1980

Programming With Recursion

Dirk Siefkes

Report Number:
80-331

PROGRAMMING WITH RECURSION

Dirk Siefkes*

Technische Universität Berlin

Inst. Software Theor. Informatik

1 Berlin 10, West Germany

February, 1980

Purdue University

Tech. Report CSD-TR 331

Abstract: Data structures defined as term algebras and programs built from recursive definitions complement each other. Computing in such surroundings guides us into writing simple programs with a clear semantics and performing a rigorous cost analysis on appropriate data structures. In this paper, we present a programming language so perceived, investigate some basic principles of cost analysis through it, and reflect on the meaning of programs and computing.

*Visiting at the Computer Science Department of Purdue University with the help of a grant of the Stiftung Volkswagenwerk. I am grateful for the support of both, VW-Stiftung and Purdue.

CONTENTS

0. INTRODUCTION

Despite the long practice in, and the broad theory about, computing, nobody seems to know what computing is. In practical programming, high-level languages have established themselves as appropriate tools for 'structured programming'. But just what is a high-level language, how do I choose the right one for my task, and where will my program be executed effeciently? In the theory of computation the competition between models of computing is fiercer still, since machine languages are also engaged. In both areas correctness proofs are done either informally or overformally, neither way showing what a program means. The situation is worse when one comes from understanding programs to writing efficient ones. In practice, everybody measures cost with his own bendable yardstick, independent of his language, and thus undependable. In theory, Turing machines and register machines have partitioned the world into an abstractly complex realm and an arithmetically complex realm, the latter containing networks and straight-line programs which are not even algorithms. In this way complexity theory buries itself under a heap of machine code which is worthless in practice, and misguiding in theory.

The objects of computation have to be created by finitistic means. In an abstract setting the Semi-Thue-systems of Post, also called 'formal grammars', serve this purpose. In practical problems one deals with functions; thus one specializes the grammar to produce terms: expressions built from finitely many primitive expressions by finitely many function symbols. Usually the representations are not unique; thus one factors the term algebra by (finitely many) equations. This way of describing a data structure by a Semi-Thue-system is often called 'algebraic specification'.

Computing then means: handle the data with the tools which are used to

construct them. Combine functions in series or in parallel, test whether a datum is primitive or compound. These are local principles, in contrast to the global principle of definition by recursion. Recursion corresponds to the inductive definition of the term algebra; explicit definition is included as a special case.

These considerations lead to the computation formalism REC which we present in this paper. Its programs are sets of recursive equations, as used by Gödel to define the recursive functions. In contrast to Gödel's definition, however, REC-programs are formatted for computing in a natural and (up to parallelism) deterministic way. Thus there is no need to define the semantics of recursive equations syntactically, by a derivation system, as following Gödel it is usually done.

REC-programs contain no data variables. Built up from the simple constructs mentioned they describe functions directly; thus they are easy to understand. This means technically that it is enough to translate the basic constructs into functions to get a 'functional semantics', and into machine code to get an implementation; the constructions extend themselves to the whole programs. It means practically that, equipped with the appropriate data structure, REC is a programming language for practical use; and, again with the right data structure, REC is easily implemented. We indicate, for example, compilers to and from an ALGOL-language. Also we construct compilers to and from Turing machines which increase the running time at most quadratically.

The approach also pays off for him who looks for cheap programs. Since the semantics is clear, it leaves no choice for the cost definition: cost traces the semantics. The functional semantics reduces the cost of a program to that of its basic constructs. The cost can be related to machine time or space, or

any other model, and thus be made as realistic of aloof as one wants. Arithmetic and machine complexity are shaped within the same language, by choosing the right data structures.

The simplicity in structure and the lack of variables make explicit the control and the data flow in the evaluation of a REC-program. Therefore in REC we distinguish 'calculation' and 'organization', and their respective costs. We define 'calculation term' and 'organization sequence', which actually do the corresponding tasks in an evaluation. We distinguish types of REC-programs, and discuss their cost analysis.

This is a technical paper; it reports on the work done so far with REC, without elaborating on variant formalisms and without reflecting on consequences. Some of the more polemic remarks in this introduction are (hopefully well-) founded by the discussion in Sect. 6. We will pursue those issues in a separate paper.

Acknowledgment: REC evolved in a series of lectures on "Theory of Computability" and "Theory of Algorithms" at the Technische Universität Berlin together with W. Fey, K. Fleischmann, B. Mahr, and F. Müller; see the lecture notes [Fleischmann - Müller - Siefkes 1975] and [Fleischmann - Mahr - Siefkes 1977]. A short account appeared in [Siefkes 1979]. I thank the people in Berlin and at Purdue for important feedback and backup by listening, arguing, and letting me go. I am especially indebted to F. Müller, who invented a first version of RECS with restricted recursion; to B. Mahr, who insisted on cost analysis and thus helped to create Sect. 4 & 5 of this paper; to J.R. Büchi, to whom I owe many of the observations and most of the aim of Sect. 6, esp. the cruical passages in 6.5; and to J. Wojcik who made me aware of the language of roots and trees.

1. THE FORMALISM REC

The language REC is a shell which grows into a programming language if one plants it over a data structure according to one's wishes. As an example for such a construction, along with REC itself we introduce and discuss an embedding into the simplist infinite data structure, the natural numbers in many representation. "Simple REC" is universal, i.e. good to define all recursive functions.

1.1 Consider the following recursive definitions of addition and multiplication over the natural numbers, using successor and predecessor:

ADD(X,Y) := if Y=o then X else ADD(X,Y-1)+1

MULT(X,Y) := if Y=o then o else ADD(MULT(X,Y-1),X)

Each line could be the body of a function procedure declaration in an ALGOL-like program. For any input x,y∈N the procedure computes an output z∈N, using an obvious evaluation strategy; MULT calls ADD as a subroutine. In this way the procedure defines a function, e.g. add: NxN → N. Let this be our motivation for REC. REC-programs are sequences of procedure definitions, which are built up from basic functions with the help of operators. Each procedure names (describes, defines) the function it computes. In computer programs variables control the data flow; procedures use variables for communication. In REC we use "data manipulating functions" instead. REC-programs thus contain variables only for functions, not for data. In this way data transfer is made explicit, and can be considered in the cost analysis.

1.2 REC-programs name partial functions $f:D^n \to D^m$ over some domain D, $n,m \geq o$. We call D^n and D^m the source and the target of f, resp. n = IDim f and m = ODim f are the input and output dimension of f. Dom f and range f are the domain of definition and range of values of f. We assume that all domains

contain an element o.

As basic functions for REC we use the one-dimensional identity, an eraser, and a zero creator; for "simple" REC further successor and predecessor, thus allowing (sequence of) natural numbers as data structure.

id:	IDim = ODim = 1	id x:= x
erase:	IDim = 1, ODim = o	erase x := ε (empty sequence)
zero:	IDim = o, ODim = 1	zero ε := o
succ:	IDim = ODim = 1	succ x := x+1
pred:	IDim = ODim = 1	pred x := $\begin{cases} x-1; & x \neq o \\ o; & x=0 \end{cases}$

The (control) operators in REC map functions into functions. An application of an operator requires matching I/O-dimensions of the arguments, the result has a fixed I/O-dimension.

composition $(f \circ g)$, IDim f = ODim g

$$(f \circ g)x := \begin{cases} f(gx); & x \varepsilon \text{Dom } g \\ \text{undefined; otherwise} \end{cases}$$

product $(f \otimes g)$,

$$(f \otimes g)(x,y) := \begin{cases} (fx, gy); & x \varepsilon \text{Dom } f,\ y \varepsilon \text{Dom } g \\ \text{undefined; o.w.} \end{cases}$$

combination (f,g) IDim f = IDim g

$$(f,g)x := \begin{cases} (fx, gx); & x \varepsilon \text{Dom } f,\ x \varepsilon \text{Dom } g \\ \text{undefined; o.w.} \end{cases}$$

test if f then g else h,

IDim f = IDim g = IDim h,

ODim f = 1, ODim g = ODim h

$$(\underline{\text{if}}\ f\ \underline{\text{then}}\ g\ \underline{\text{else}}\ h)x := \begin{cases} gx; & fx = o \\ hx; & fx \neq o \\ \text{undefined}; & x \notin \text{dom } f \end{cases}$$

Later we will define the following auxiliary function in REC:

$\underline{\text{selectors}}\ \ s^n_{j1,\ldots,jm}, 1 \leq j_i \leq n, \quad \text{IDim} = n, \text{ODim} = m$

$s^n_{j1,\ldots,jm}(x_1,\ldots,x_n) := (x_{j1},\ldots,x_{jm});$

especially the $\underline{\text{projections}}\ s^n_j;$

$\underline{\text{constant functions}}\ c^n_k, \quad \text{IDim} = n, \text{ODim} = 1$

$c^n_k x := k.$

1.3 Definition: Syntax of the language REC(B)

Let B be a set of function symbols with fixed I/O-dimensions (function constants). For $i,n,m \in \mathbb{N}$ let $F^{n,m}_i$ be a function variable of IDim n and ODim m.

(a) Terms: Any function variable and any function constant ID, ERASE, ZERO, ZERO, or G where $G \in B$, is a term. If u,s,t are terms of suitable I/O-dimensions, so are $(s \circ t)$, $(s \circledast t)$, (s,t), and (if s then t else u). Note that any term has fixed I/O-dimensions.

(b) Programs: For $i=1,\ldots,k$ let F_i be different function variables and t_i terms with I/O-Dim t_i = I/O-Dim F_i. If the terms contain no other function variables, then

$$(F_1 := t_1; \ldots; F_k := t_k)$$

is a program with the I/O dimension of F_1.

(c) To simplify notation we suppress indices and brackets wherever possible, and use expressions like ADD, or ADDFIRSTo for function variables.

(d) As an example we rewrite the definition of multiplication in 1.1 as a REC(B)-program with $B=\{S_1^2, S_2^2, C_o^2, SUCC, PRED\}$:

$$MULT := \underline{\text{if}}\ S_2^2\ \underline{\text{then}}\ C_o^2\ \underline{\text{else}}\ ADD\circ(MULT\circ(ID\otimes PRED), S_1^2)$$

$$ADD := \underline{\text{if}}\ S_2^2\ \underline{\text{then}}\ S_1^2\ \underline{\text{else}}\ SUCC\circ(ADD\circ(ID\otimes PRED))$$

1.4 The semantics of a REC-program will be the function computed by it. The evaluation of a REC-program on an input consists of a sequence of steps of the following three types: (i) evaluation of function constants for given arguments, (ii) manipulation of data through control operators, (iii) (possibly recursive) calls for the evaluation of function variables on given arguments using their definition. (This is a call-by-value strategy: we transfer data only, and do not substitute terms.) For example, the evaluation of the multiplication program in 1.3.d yields the following steps:

$MULT(2,2) = t_1(2,2) =$ $\quad =4$

$S_2^2(2,2) = 2$

$= ADD\circ(MULT\circ(ID\otimes PRED), S_1^2)(2,2) =$ $\quad =4$

$(MULT\circ(ID\otimes PRED), S_1^2)(2,2) =$ $\quad =(2,2)$

$MULT\circ(ID\otimes PRED)(2,2) =$ $\quad =2$, $S_1(2,2) = 2$

$(ID\otimes PRED)(2,2) =$ $\quad =(2,1)$

$ID(2) = 2,\ PRED(2) = 1$

$= MULT(2,1) = \ldots$

$= ADD\circ(MULT\circ(ID\otimes PRED), S_1^2)(2,1)$

$\ldots MULT(2,0) = \ldots = 0 \ldots$

$= ADD(0,2) = \ldots = 2$

$= ADD(2,2) = \ldots = 4$

1.5 Definition: Semantics of REC(B)

For each constant G∈B fix a function g with the same I/O-dimensions. g is the interpretation of G. B, or the set {g;G∈B}, is the basis of REC(B).

We define the function

VAL:programs → [terms → functions]

which describes the evaluation of REC(B)-programs. Let

$A := (F_1 := t_1, \ldots, F_k := t_k)$

be a REC(B)-program. The semantics of A is the function $VAL(A)(F_1)$. We define VAL(A) by recursion over the structure of terms. (We drop the argument A. For simplicity we assume that all arguments have the correct dimension; for others VAL(t) is not defined. Expressions with undefined subexpressions are undefined.)

(1) $VAL(p \circ q)b := VAL(p)(VAL(q)b)$

(2) $VAL(p \otimes q)(b,c) := (VAL(p)b, VAL(q)c)$

(3) $VAL(p,q)b := (VAL(p)b, VAL(q)b)$

(4) VAL (if a then p else q)b :=
- VAL(p)b; VAL(s)b=o
- VAL(q)b; VAL(s)b≠o
- undefined; o.w.

(5) VAL(ID)b := b, VAL(ERASE)b := ε

VAL(ZERO)ε := o, VAL(G)b := gb for G∈B

(6) $VAL(F_i)b := VAL(t_i)b$

(7) VAL(t)b is undefined if its computation does not terminate.

1.6. The definition of the evaluation function VAL is itself recursive. The evaluation of VAL, however, is unique except for the parallel calls in clauses (2) + (3). As proved in 6.1+2, either order (as well as parallel computation) will yield the same result in the end, since (1) - (6) is strictly call-by-value. Thus one need not to worry about what evaluation strategy to choose for VAL. In this sense def. 1.5 yields a deterministic evaluation strategy for any REC-program. (See also 3.6)

One gets REC-programs from the sets of equations used by Gödel to define recursive functions (see [Gödel 1934]), by

(i) allowing arbitrary output dimension;

(ii) permitting only (different) function variables as terms on the left side;

(iii) making the evaluation strategy deterministic;

(iv) eliminating the number variables.

In the following definition, we therefore call functions on the natural numbers 'recursive' if they are computable by an RECS-program. Sect. 3 shows that the two definitions actually are equivalent. - We will discuss the issue of semantics in more detail in Sect. 6.

1.7 Definition:

(a) A program computes, or defines, or names its semantics. Two programs with the same semantics are equivalent.

(b) RECS, or simple REC, is REC over the basis {succ,pred}. A function is called recursive if it is computable by an RECS-program.

(c) We write REC also short for REC(B), if the basis is of no concern, or is clear from the context. On the other hand, we may include the function constants id, erase, zero into the basis.

1.8 Lemma: The selectors and constant functions of 1.7 are recursive, and so are addition, multiplication and exponentiation.

Proof: A REC-program for the projection s_j^n is

$$S_j^n := \underbrace{\text{ERASE}\otimes\ldots\otimes\text{ERASE}}_{j-1}\otimes\text{ID}\otimes\underbrace{\text{ERASE}\otimes\ldots\otimes\text{ERASE}}_{n-j}$$

This yields REC-programs for the selectors:

$$S^n_{j1,\ldots,jm} := (S^n_{j1},\ldots,S^n_{jm})$$

(By 1.5 the operators ⊛, (,) and ∘ are associative. This helps to save brackets.) We build up RECS-programs for the constant functions inductively:

$$C^o_o := \text{ZERO},\ C^o_{k+1} := \text{SUCC}\circ C^o_k,\ C^{n+1}_k := C^n_k \circledast \text{ERASE}$$

Section 1.3 contains RECS-programs for addition and multiplication; esponentiation works similarly. Q.E.D

1.9 The programs in the proof of 1.8 contain function symbols which are defined through earlier programs. We get complete REC-programs by adding the corresponding lines. In this sense we will use function symbols in REC-programs, especially for selectors and constant functions.

The function constants of REC, id, erase, and zero, together with the contol operators serve to manipulate data; the basic functions, e.g. succ and pred in RECS, change data. This distinction will be important for the cost analysis. To allow for easier writing and reading of REC-programs, however, we will allow variables for (sequences of) natural numbers, although they obscure the data flow. The programs for addition and multiplication show how this transition works. Formally we introduce a new language: REC with variables.

1.10 Definition:

(a) We introduce variables and constants for numbers into a REC-program in the following way: Change each line F:=t into FX:=tx, using as many variables as the input dimension is. Then "evaluate" the terms for these variables instead for

data. To this end we change the procedure VAL of 1.5 into VAR, altering (4) and (6) as well into:

(4) VAR(if s then p else q) x := if VAR(s)x then VAR(p)x else VAR(q)x

(6) VAR(F) x := Fx,

and replacing (7) by clauses for selectors and constant functions:

(7) $VAR(S^n_{j1,\ldots jm})x := (x_{j1},\ldots,x_{jm})$

$VAR(C^n_k)x := k$

The semantics of a REC-program with variables is obvious from 1.5.

b) Let Fx:=t be a function procedure where the term t contains no other variables than $x=(x_1,\ldots,x_n)$. We eliminate variables and constants for numbers from this line in the following way: Cancel x behind F. Within t replace each occurrence $(s_{j1},\ldots,x_{jm})$ of variables by the selector $S^n_{j1,\ldots,jm}$ and each number constant k by the symbol C^n_k. Finally change each application s(t) into a composition s∘t. The result of this procedure, applied to a sequence A of lines, is called DEVAR(A). We call A a REC-program with variables if DEVAR(A) is a correct REC-program.

1.11 Theorem: For any REC-program A and any REC-program B with variables: A and VAR(A) are equivalent, and so are B and DEVAR(B).

Proof: By induction on the program structure. Q.E.D.

1.12 From now on we weaken our requirements about REC-programs: we allow variables for numbers, and identify functions and function symbols, using the same expressions for both.

2. EXTENSIONS AND VARIATIONS

The language RECS is good to define all computable functions over the natural numbers. (See Sect. 3.) We can handle other problems through RECS by coding their data structures into $\mathbb{N}$. Instead we will modify REC for two more complicated data structures. Here are two reasons for doing so: (i) All problems come with their peculiar data structures. Only in keeping it, we can write good programs and analyze their cost. (ii) We facilitate the comparison of REC with other formalisms, say ALGOL or Turing machines. Especially, we can write compilers which do not change the cost of programs too much. RECS has all the computation power we can wish for, but it is a poor language for writing efficient programs.

Most data structures, like matrices, trees, or any graphs, can be (and are in praxis) easily represented by lists (finite sequences). Since Turing machines, too, work on linear input, we will restrict ourselves to this type. A function on lists can be thought of as having either one argument of variable length (word) or a variable number of arguments (sequence). Trivially a word is a finite sequence; again a finite sequence can be coded into a word, using an extra symbol. Regarding the data access, however, both structures are quite different. In 2.3-6, we will therefore consider REC over words; in 2.7-9, we will modify REC to compute functions with a variable number of arguments. We start in 2.1+2 with the general notions of 'language' and 'compiler'.

2.1 Definition:

(a) An (abstract algorithmic) language is a pair (L,I) where

(i) L is a decidable set, called programs:

(ii) I is a function on L, called interpretation or semantics, which to any program yields a computable function.

(b) (M,J) is an extension of (L,I) if $L \overset{c}{\subseteq} M$ and $J \upharpoonright L = I$.

(c) A compiler from (L,I) to (M,J) is a computable function $C: L \to M$ s.t. $J \bullet C = I$.

(d) Two languages are equivalent if they admit compilers in both directions.

2.2 Remarks:

(a) If (M,J) is an extension of (L,I), then the inclusion is a trivial compiler. Both languages are equivalent, if there is a compiler in the other direction as well.

(b) If there is a compiler from (L,I) to (M,J), then $\text{range}(I) \overset{c}{=} \text{range}(J)$. Thus equivalent languages compute the same functions.

(c) If $B \overset{c}{=} B'$, then REC(B') is an extension of REC(B). If B contains succ and pred and only recursive functions, then REC(B) is equivalent to RECS. REC(B) with and without variables are equivalent through the compilers VAR and DEVAR of 1.10.

2.3 We will now collect the concepts we need to apply REC to word functions. For any finite set M let M^* be the free monoid over M. A member $m_1 \ldots m_k$ of M^* is called a word, of length k. The unit of M^* is the empty word ε, of length o. The monoid operation is concatenation

$$\text{cat}(m_1 \ldots m_k, n_1 \ldots n_h) := m_1 \ldots m_k n_1 \ldots n_h.$$

We consider the following operations on M^* (assume $o \in M$):

$$\text{equal}(v,w) := \begin{cases} o; & v,w \in M,\ v=w \\ oo; & o.w. \end{cases} \quad \text{(equal letter)}$$

$$\text{first } w := \begin{cases} m; & w=mv,\ m \in M \\ oo; & w=\varepsilon \end{cases} \quad \text{(first letter)}$$

$$\text{subfirst } w := \begin{cases} v; & w=mv,\ m \in M \\ oo; & w=\varepsilon \end{cases} \quad \text{(substract first letter)}$$

$$\text{addfirstm } w := mw \text{ for } m \in M \quad \text{(add first letter m)}$$

The corresponding last-operations are defined analogously. For 'REC over words' (def. 2.4) we will use as data structure M*, not the free monoid, but the algebra generated (not freely) from the empty word by the "successor functions" addfirstm and addlastm, and further equipped with their converses first, last, subfirst, sublast, and with the test for equal letters. Now let $q \in \mathbb{N}$, $q \geq 1$. Write q short for $\{0,\ldots,q-1\}$. Let $N_q \overset{c}{=} q^*$ be the set of q-ary representations of the natural numbers. Any function on $\mathbb{N}$ can be translated into a function on N_q, and then extended to q^* by identifying ow with w for $w \neq \varepsilon$. Conversely, for q>1 addfirst1 is a natural one-to-one coding of q^* into N_q, which serves to translate functions over q^* into functions over $\mathbb{N}$.

2.4 Definition: For $q \geq 1$ let

$$B_q := \{EQUAL, FIRST, LAST, SUBFIRST, SUBLAST, ADDFIRSTm, ADDLASTm; m<q\}$$

with interpretations over q^* as in 2.3. Let RECWq := $REC(B_q)$ be the language REC for q-ary wordfunctions (REC over words for short). By 2.3 we can use RECWq to compute functions over $\mathbb{N}$; normally we will use RECW := RECW2.

2.5 Examples:

a) The word functions cat and length are computable in RECWq, further

$$\text{empty } w := \begin{cases} o; & w=\varepsilon \\ oo; & \text{o.w.} \end{cases} \quad \text{(empty word)}$$

$$\text{eq } (u,v) := \begin{cases} o; & u=v \\ oo; & \text{o.w.} \end{cases} \quad \text{(equality)}$$

$$\text{gaddfirst}(u,v) := \begin{cases} uv; & u \in q \\ \varepsilon; & \text{o.w.} \end{cases} \quad \text{(generalized addfirst)}$$

$$\text{firstm } w := \begin{cases} u; & w=umv, \text{ m not in u} \\ w; & \text{o.w.} \end{cases} \quad \text{(part before first m, } m \in q\text{)}$$

$$\text{subfirstm } w := \begin{cases} v;\ w=umv,\ m \text{ not in } u \\ w;\ o.w \\ (\underline{\text{subtract first}}\ m\ \underline{\text{and part before}},\ meq) \end{cases}$$

and the analogous last-functions.

b) The functions successor and predecessor are computable in RECW.

Proof:

a) We present RECWq-programs. (Note that we use o for TRUE, and oo for FALSE.)

```
empty u : = if addfirsto(u)=o then o else oo

eq(u,v) := if empty(u) then (if empty v then o else oo)
           else if empty(v) then oo else if equal(first u, first v)
                then eq(subfirst u, subfirst v) else oo

gaddfirst(u,v) := if equal(u,o) then addfirsto(v) else
                  .
                  .
                  .
                  if equal(u,q-1) then addfirst(q-1)v else ε
```

where ε := subfirst(o), and $o := c_o^n$ (n depends on the program)

```
cat(u,v) := if empty(u) then v else cat(sublast(u), gaddfirst(last (u),v))

length w := if empty(w) then o else succ(length(sublast w))
```

We leave the search functions as an exercise.

b) The successor of the binary number uol...1 is ulo...o. Thus a RECW program for the successor function is:

```
succ w :=  if empty w then 1 else
           if first w = o then succ(subfirst w) else
           if last w = o then addlast1(sublast w)
           else addlasto(succ(sublast w))
```

The program for pred and the analogous programs in RECWq are left as an exercise. If one uses the q-ary successor in a), one gets the length as a q-ary number. Q.E.D.

2.6 Remark: If we identify functions over $\mathbb{N}$ and functions over N_2, then we get a compiler from RECS into RECW by adding the programs for succ and pred of 2.5b to RECS-programs. This compiler is of no use, since the compiled programs do not utilize the structure of N_2.

2.7 RECW is good for programming such functions as bitwise addition and multiplication. Functions on, e.g. matrices or graphs,require direct access to all parts of the input. We regard the elements of such lists as separate arguments.

For any set D s.t.o$\in$D, and any e$\notin D^*$ let E := E(D) := $D^* \cup \{e\}$. For any n$\geq$o we embed D^n into E. We will regard partial functions on D^q or D^* as functions with E as source and target; we will use e for error handling (dimensions do not fit). (Following D. Scott one can consider E as a lattice with e as top and 'undefined' as bottom. If one misuses"undefined" for error handling, as we did in Sect.1, one has a partial order. In practical programming, however, one would distinguish the message "dimension error" from non-termination, which results in "time out".) For a$\in$E we define the dimension as

$$\dim a := \begin{cases} \text{length } a; & a \in D^* \\ e; & a = e \end{cases}$$

We require fe=e for any function f on E. We extend a function $f:D^n \to D^m$ to $f^*:E \to E$ by

$$f^* a := \begin{cases} fa; & \dim a = \text{IDim } f \\ e; & \text{o.w} \end{cases}$$

f and f^* thus coincide on source f, especially dom f = (domf^*)$\restriction$(source f). We define I/O-Dim f^* := I/O-Dim f. Normally we will then write f instead of f^*. Any function which is not an extension in this sense, is of variable I/O-dimension. For such an f the equations I/O-Dim(f)=n shall be true for any n$\in\mathbb{N}$.

For REC over E we need two more basic functions:

the variable identity vid, I/O-Dim variable,

vid x := x;

the variable eraser verase, IDim variable, ODim=o,

$$\text{verase } x := \begin{cases} \varepsilon; & x \varepsilon D^* \\ e; & x=e \end{cases}$$

For easier writing we put

$$(x_1,\ldots,x_n) := \begin{cases} e; & \text{if } x_i=e \text{ for some } i \\ \text{undefined}; & \text{o.w. if } x_i \text{ undefined for some } i \end{cases}$$

Then the definition of composition and combination of 1.2 carries over to E unchanged. The test operator has to pass on error messages:

$$(\underline{\text{if}}\ f\ \underline{\text{then}}\ g\ \underline{\text{else}}\ h)\ x := \begin{cases} e; & fx=e \\ gx; & fx=o \\ hx; & fx \neq o \\ \text{undefined}; & \text{o.w.} \end{cases}$$

For the product we distinguish grouping from left and from right. In $(f@g)x$ thus f takes as much of x as it can digest (i.e. all of x if IDim(f) is variable; otherwise IDim(f) many arguments if there are); g gets the rest. Let $x \varepsilon D^*$, dim(x)=n :

$$(f@g)x := \begin{cases} (f(x_1,\ldots,x_j),g(x_{j+1},\ldots,x_n)); & \text{where } j:=\max\{i;\ o \leq i \leq n, \\ & \qquad \text{IDim}(f)=i,\ \text{IDim}(g)=n-i\} \\ e; & \text{if no such i exists} \end{cases}$$

$f\hat{@}g$ is defined analogously from the right, using "min" instead of "max".

2.8 Definition: Let the domains D and E be as in 2.7, let B be a set of functions over E. The language VREC(B) for functions over D with variable dimension is defined as follows: Function constants are id, erase, zero extended to E, further vid, verase from 2.7, and the functions in B; control

operators are composition, left and right product, combination, and test-on-zero from 2.7; syntax and semantics are as for REC. We identify functions and control operators with their extensions, and thus regard VREC(B) as an extension of REC(B). Especially we are interested in VRECS (simple VREC) and VRECW (VREC for word functions).

2.9 Examples:

a) The data access functions which correspond to the word functions in 2.3÷5, are computable in VREC. (Of course there is no analogon to concatenation.)

b) Variable-length addition and multiplication, and the dimension function are computable in VRECS.

Proof:

a) We give a few examples. For $x \in E$ let

$$\text{vfirst } x := \begin{cases} x_1; & \dim \geq 1 \\ e; & \text{o.w.} \end{cases}$$

$$\text{vsubfirst } x := \begin{cases} (x_2,\ldots,x_n); & n:=\dim(x)\geq 1 \\ e; & \text{o.w.} \end{cases}$$

$$\text{vaddfirsto } x := (o,x)$$

Then VREK-programs are

vfirst := (id&verase)

vsubfirst := (erase&vid)

vaddfirsto := (zero&vid)

b) The function $\sum_{i=1}^{n} x_i$ is defined by the program

VADD := if VADDFIRSTo then ZERO else ADD∘(ID&VADD)

Finally

DIM := if VADDFIRSTo then ZERO else SUCC∘DIM∘VSUBFIRST

2.10 The reader should have no problems to compile the REC-formalisms considered into his favorite programming language. To translate a REC-program A, for example, into an ALGOL-like language, first put variables into A using 1.10. Then change any line Fx := t into a procedure declaration

function F(X); F := t

and add ALGOL-procedures for the functions and operations used in A. For example, in ALGOL one computes (f⊗g)(a,b) by computing fa and gb, and combining the results into a field [fa,gb]. Finally one gets an ALGOL-program equivalent to A by declaring the types of variables, and specifying input and output.

2.11 Let us discuss briefly other types of REC-formalisms.

a) As mentioned in 1.6 the definition of the semantics of REC does not specify in which order (f⊗g)b and (f,g)b are to be evaluated. Depending on the computation facilities, one can implement VAL in such a way that (possibly bounded) parallel computation is possible. This might create synchronization problems. Call this version PARREC.

b) We can extend REC to NREC by adding a function constant COIN, of IDim 0 and ODIM 1, which takes the values 0 and 1 (or some other value ≠ 0) nondeterministically. Thus NREC allows nondeterministic computations. Conveniently, COIN is used as a condition in if-then-else-terms only. The semantics definition 1.5 is extended by adding the nondeterministic line

VAL(COIN)ε := 0 or VAL(COIN)ε := 1.

Thus now the semantics of a program is a relation.

c) Similarly we can add function constants COINp, where $0 \leq p \leq 1$, with the semantics

VAL(COINp)ε := 0 or VAL(COINp)ε := 1 with probability p.

This formalism, call it PROBREC, allows to model the types of probabilistic algorithms which are used in the literature. For example, the term

if COIN(1/3) then f else if COIN(1/2) then g else h

allows to choose with equal probabibility between the computation of f, g, and h. The semantics of a program in PROBREC is again a relation, but now with probabilities for the values.

d) In [Kirchner-Röhrich-Siefkes 1979] we use a logic language SYL to synchronize the computations of nondeterministic procedures working in parallel. SYL lends itself to be combined with REC to allow the types of computation indicated above. By intertwining the formal semantics of REC and SYL we get a powerful and convenient formalism to specify "synchronized computation systems", a quite general model of asynchronous computations.

3. THE EXPRESSIVE POWER OF REC

In this section we will do some programming in REC. We will introduce the use of predicates in REC-programs, and sketch a compiler from a restricted ALGOL-language into REC. To prove that REC is universal we will give easy compilers between REC and multi-tape Turing machines. Since this type of development is familiar from traditional formalisms, we can be rather brief here.

3.1 As usual we call a predicate recursive if its characteristic function is recursive. Contrary to normal usage we allow partially defined predicates, since we are interested in conditions on partial functions, e.g. fx=0. As implicit in example 2.5, we define the characteristic function cp of a predicate p as

$$cp\ x := \begin{cases} 0 \text{ ; px true} \\ I \text{ ; px false} \\ \text{undefined ; px undefined} \end{cases}$$

We generalize REC-programs by allowing predicates in the place of their characteristic functions. Part a) of the following lemma together with lemma 1.8, furnishes a proof that recursive predicates are closed under Boolean operations and bounded quantifiers, which means:
If p,q are recursive predicates, so are $\neg p$, $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \leftrightarrow q$, $(\exists x \leq y)p$, $(\forall x \leq y)p$.

3.2 Lemma: a) The functions signum and anti-signum

sg x := if x=0 then 0 else 1

$\overline{sg}$ x := if x=0 then 1 else 0

are recursive, and so are iterated sum and product

$$sumg(y,z) := \sum_{x=0}^{y} g(x,z), \quad prodg(y,z) := \prod_{x=0}^{y} g(x,z),$$

if g is a recursive function.

b) Subtraction and integer division

$$x\dot{-}y := \begin{cases} x-y & ; y \leq x \\ 0 & ; y > x \end{cases}, \quad \left[\frac{x}{y}\right] := \begin{cases} z & ; y \neq 0 \wedge z \leq \frac{x}{y} < z+1 \\ \text{undefined} & ; \text{o.w.} \end{cases}$$

are recursive, and so are the predicates $=, \leq, <$.

c) The generalized projections with variable IDimension

$$vs(i,x) := \begin{cases} x_i & ; 1 \leq i \leq \dim x \\ e & ; \text{o.w.} \end{cases}$$

are computable in VREC. They are needed in VREC for data access.

Proof: We provide the necessary REC-programs.

a) The definitions of sg and $\overline{sg}$ are REC-programs.

sumg(y,z) := if y=0 then g(0,z) else sumg(y$\dot{-}$1, z) + g(y,z) ;

prodg is analoguous.

b) $x\dot{-}y$:= if y=0 then x else pred(x)$\dot{-}$pred(y)

$x \leq y$:= if x=0 then 0 else if y=0 then 1 else pred(x)$\leq$pred(y)

$x=y := x \leq y \wedge y \leq x$; $x < y := \neg\, y \leq x$

$\lfloor x/y \rfloor$:= if $x < y$ then 0 else $\lfloor x\dot{-}y/y \rfloor + 1$

c) $vs(i,x_1,\ldots,x_n)$ = if i=0 $\vee$ n=0 then e else

if i=1 then x_1 else $vs(i\dot{-}1, x_2,\ldots,x_n)$

This translates into the VREC-program (see example 2.9a)

vs := if vfirst=0 then e else if vfirst=1 then vfirst$\circ$vsubfirst

else vs$\circ$(pred@vsubfirst)

where e stands short for any term producing an error-message; as zero.

(Check how the cases n=0 and i > n are handled.) Q.E.D.

3.3 Continuing in the spirit of 3.1+2 we could build up now a library of REC-programs and programming techniques, which would enable us to use REC like any programming language. Instead we will show that REC is equivalent to an ALGOL-language. This way also one can transfer our treatment of cost analysis in Sect. 4 & 5 to other languages.

The language we think of has the concepts of ALGOL 60, except goto and switch. As data we allow natural numbers or words, single or in arrays; also Boolean values, which we identify with 0 and 1. To translate such a program into REC do the following:

1. step: Determine the REC-language to be used : REC, VREC, RECW, VRECW; then cancel the declarations of variables. Determine the function basis B by collecting all undefined functions and predicates.

2. step: Working inside out with the help of lemma 3.4 replace all while- and for-loops by recursive procedures. Also program other control structures in REC.

3. step: Change procedure bodies into REC-lines, chopping off the heads. Use procedure names as function variables, and local variables (i.e. variables used only inside the procedure) as data variables. Global variables (used in more than one procedure) have to be made function variables, too. The assignment statements inside a procedure are to be worked into the term(s) of the corresponding REC-line(s). Use auxiliary function variables freely.

4. step: Use the input/output specifications of the ALGOL-program to determine the first line of the REC-program; then omit them.

5. step: If wanted, eliminate the data variables with the help of 1.10.

This compiler is admittedly vague. A more detailed presentation, however, would not only be outside, but against the scope of this paper : programs from a machine-oriented language like ALGOL will not become better structured by translating them into a function-oriented language like REC.

3.4 Lemma: In ALGOL 60, while- and for-loops can be replaced by recursive procedures.

Proof: This is of course a standard fact. We indicate the construction to make the presentation of the compiler self-contained.

a) The procedure

```
procedure H(X);
begin H := F(X); while P(X,H) do H := G(X,H) end
```

can be replaced by the following equivalent pair of procedures

```
procedure H(X); H := H̄ (X,F(X))
procedure H̄(X); H̄ := if ¬P(X,Y) then Y else H̄(X,G(X,Y))
```

b) The body of the procedure

```
procedure  H(X,U,V,W);
begin H := F(X); for I=U step V until W do H := G(X,I,H) end
```

is equivalent to

```
begin H := F(X); I := U; while I ≤ W do
     begin H := G(X,I,H); I := I+V end  end
```

Thus as in a) the whole procedure can be replaced by

```
procedure H(X,U,V,W);  H := H̄(X,U,V,W,F(X))
procedure H̄(X,U,V,W,Y);
H̄ := if U > W then Y else H̄(X,U+V,V,W,G(X,U,Y))
```

Note that in b) the pair (H,I) (or (Y,U) in $\overline{H}$) plays the same rôle as H (or Y in $\overline{H}$) in a). Q.E.D.

3.5 From now on we will assume that any 'computable' function can be programmed in a REC-formalism over the appropriate data structure. Especially, the different REC-formalisms are all equivalent if we code and decode the data correspondingly. For example, we can translate a RECW-program multiplying integers bitwise, into an RECS-program; similarly for a VREC-program finding

Hamiltonian circuits in a graph, by coding graphs into sequences, sequences into words, words into numbers. As a less practical and more precise way of proving the universality of REC, in the rest of the Section we will present compilers between Turing machines and REC-programs. We will be more detailed here, since in Sect. 4 we will investigate the cost increase of these compilers.

3.6 For convenience we consider multi-tape Turing machines with two special tapes for input and output, resp.; we need not to be specific about details. We assume machines to be programmed in a language TURING, and identity programs and machines. Let $A \equiv (F_1 := t_1,\ldots,F_\ell := t_\ell)$ be a program in any REC(B)-formalism where B is a finite set of computable functions. We construct a Turing machine M which simulates the evaluation of A, and thus is equivalent to A. M has two working tapes, and has the program A, as well as programs for the functions in B, built into the hardware. Started with an input b, M prints $F_1 b$ on tape 1. Using tape 1 as a stack and tape 2 for the calculation of basic functions, M now evaluates $F_1 b$, following the definition 1.5 of VAL. At each stage of the evaluation, tape 1 contains a sequence $t_1;\ldots;t_m$ where each word t_i is either

(i) a <u>subterm</u> of A, or

(ii) a <u>datum</u>, i.e. a possible input (sequence), or

(iii) an <u>evaluation term</u> pc, i.e. p is a subterm of A and c is a datum, or

(iv) a pair [pc,qd] where each element is an evaluation term or the empty word.

The following table says what M does depending on whether t_m is an evaluation term (lines 1-6; corresponding to 1-6 in the definition of VAL) or a "returned value" (lines 7-10). M does line 6 by getting t_i from the internally stored program.

top of stack		changed into
(1)	$(p \bullet q)c$	$p;qc$
(2)	$(p \circledcirc q)c$	$[,qc];pd$ where $c=(d,e)$
(3)	$(p,q)c$	$[,qc];pc$
(4)	(if s then p else q)c	(if s then p else q)c;sc
(5)	gc , g function constant	d , =g(c) computed on tape 2
(6)	$F_i c$	$t_i c$
(7)	p;c	pc
(8)	[,pc];d	[d,];pc
(9)	[c];d	(c,d)
(10)	(if s then p else q)c;d	{ pc ,if d=0 { qc ,if d≠0

If neither of these cases applies, then tape 1 contains a single datum. M prints this on the output tape, and halts. By induction on the number of evaluation steps we can prove that M simulates the computation of $VAL(A)(F_1)b$. (Note that we have fixed the order of evaluation for $(p \circledcirc q)$ and (p,q).) - We could change M into a universal REC-interpreter by putting the program A on the input tape instead into the hardware.

3.7 Theorem: There is a compiler from any REC-formalism with a computable basis into TURING.

3.8 To get the converse compiler let M by any Turing machine. A computation of M can be described by a (possibly infinite) sequence of configurations. Let H be the set of halting configurations of M. Let g be the function which to any configuration not in H yields the next configuration. Then the function

(1) $fx := \underline{\text{if}}\ x \in H\ \underline{\text{then}}\ x\ \underline{\text{else}}\ (f \circ g)x$

is the "configuration function computed by M", i.e. if M is started in configuration x, then fx is the halting configuration if there is any, and undefined

otherwise. The predicate εH and the function g are easily computable; thus by 3.5 they can be programmed in REC. These programs together with (1) yield a REC-program for f. (To be more specific, we can code configurations into words, and thus get a RECW-program.) If M is used to compute a function $h : D \to D$, we have

(2) $h := write \circ f \circ read$,

where for $a \varepsilon D$ read(a) is the initail configuration of M on input a, and $write(b) \varepsilon D$ is the output for the halting configuration b. Thus if we add (2) to the program for f, we get a REC-program for h.

3.9 Theorem: There is a compiler from TURING into REC.

3.10 Corollary: Every recursive function can be computed by an RECS-program of the form

f := write ∘ g ∘ read

g := if finf then id else g ∘ nextf

Here work, read, finf, and nextf (the two latter ones depending on f) are very simple functions (e.g. elementary) which manipulate bit representations of numbers. This is an analogue to Kleene's Normal Form for μ-recursive functions.

3.11 It is not hard to extend the given compilers to the formalisms mentioned in 2.11. Parallel computation in REC can be done on systems of Turing machines which are interrelated by extra heads on their input/output types: each machine can print its output onto the input tapes of the other machines. Conversely such systems can be simulated in PARREC. Similarly, nondeterministic and probabilistic Turing machines correspond to nondeterministic and probabilistic REC-programs, resp.

4. COST OF REC-PROGRAMS

To get an idea of the complexity of a problem one calculates the cost of fast algorithms solving it, and tries to contrast these "upper bounds" with "lower bounds" which no algorithm can beat. In practical cases one determines the cost of an algorithm on a given input by counting how often the "leading operation(s)" is(are) executed during the evaluation. In this rough way one estimates the total number of steps of the algorithm, and thus the time it consumes. Often one does the counting in one sweep for inputs of the same "size"; the size might be the length of a word or list, or the dimension of a matrix, or the number of nodes of a graph. The cost of the algorithm as function of the input size is then defined as the maximal cost for inputs of that size. For a concrete programming language with its many features it is hard to define "cost" precisely. We will give a definition for our REC-languages. This definition can be transferred to concrete ALGOL-like languages, and can thus serve as a basis for a rigorous cost analysis.

We start with defining "input size" and "operation cost". In REC an operation can be either a function constant or a control operator (including call). In (V)REC(B) any operation "costs" a natural number, independent of the argument. Thus we count the operations, weighted according to their difficulty. Normally, however, any function constant in B will be charged 1, the others 0; thus data manipulation (including control operators) is free. In REC over words we are less generous: we charge for data manipulation, too; the cost accounts for Turing machine steps, thus depends on the length of the argument, at least for the single-taps case (see 4.3c).

4.1 Definition: Input size and operation cost in REC-formalisms:

Let B be a set of functions, let $x=(x_1,\ldots,x_m)$ be an input, let $q\in\mathbb{N}$, $q\neq 1$:

a) REC(B) over N, esp. RECS: The size of x is its numerical value $\sum_{i=1}^{m} x_i$.

Normally the cost is o for id, erase, zero, and any operator, and 1 for f∈B (sometimes other natural numbers).

b) VREC(B): Normally, the size of x is its dimension m; otherwise a monotone function of m. The operation cost is as in REC(B), but often ≠o for some special functions only.

c) (V)RECWq(B): The size of x is its length. For all operations the cost is 1 (multi-tape cost), or the length of the arguments (single-tape cost), or else a monotone function thereof.

4.2 Definition: a) Cost of terms and programs, depending on the input:

Following in parallel the semantics definition 1.5 we define the evaluation cost of REC-programs as a recursive function

COST: programs → [terms → [inputs → $\mathbb{N}$]]

The basis of the recursion is the operation cost OPCOST(op)b of an operation op on the input b. Thus let A :≡ (F_1 := t_1,...,F_k := t_k) be a REC-program. We write COST instead of COST(A).

(1) COST(p∘q)b := COST(p)(VAL(q)b)+COST(q)b+OPCOST(∘)(VAL(q)b)

(2) COST(p⊗q) (b,c) := COST(p)b+COST(q)c+OPCOST(⊗)(b,c)

(3) COST(p,q)b := COST(p)b+COST(q)b+OPCOST(,)b

(4) COST(if s then p else q) b := $\begin{cases} m+COST(p)b; & VAL(s)b=o \\ m+COST(q)b; & VAL(s)b\neq o \\ \text{undefined}; & o.w \end{cases}$

where m := COST(s)b+OPCOST(test)b

(5) Let f be a function constant:

COST(f)b := OPCOST(f)b

(6) COST(F_i)b := COST(t_i)b+OPCOST(call)b

(7) COST(A) := COST(F_1)

b) Cost of programs, depending on the input size:

We use the same name for the size cost function

$$COST : programs \to [terms \to [\ \mathbb{N}\ \to \mathbb{N}\ ]].$$

Let A be a program, let $n \in \mathbb{N}$:

$$COST(A)(p)n := \begin{cases} \max\{COST(A)(p)b;\ size(b)=n\}\ ; \text{ if defined} \\ \text{undefined; if this set is infinite or contains an undefined cost} \end{cases}$$

$$COST(A) := COST(F_1)$$

4.5 Examples:

a) The REC-programs for the projections s_i^n in 1.8 contain only the operations id and erase; thus they cost nothing. The RECS-program for the constant function c_k^n costs k. The cost for addition and multiplication in RECS then amounts to $O(n)$ and $O(n^2)$, resp.

b) The easy VRECS-program for the access operation $vs(i,x_1,\ldots,x_n) := x_i$ in 3.2 calls itself i-1 times using pred. Thus normally data access will be free in VREC(B); in VRECS it costs $O(n)$.

c) Under single-tape cost in RECW any single step, and thus any line without calls, is of linear cost. Therefore a recursion of depth m on an input of size n costs $O(m \cdot n)$. For example, accessing and scanning data (e.g. by a selector) is of linear cost, anything involving copying (like concatenation, test for equality, or bitwise addition) is quadratic, and bitwise multiplication has cubic cost. Thus single-tape RECW-cost corresponds to time on a single-tape Turing machine (which has to run through the input to access it). Like multi-tape Turing machine time, multi-tape RECW-cost is cheaper by a factor of n. Thus data access costs $O(1)$, copying and the like is of linear cost, and bitwise multiplication is quadratic. Note that in working with natural numbers, RECW is exponentially cheaper than RECS; this is the familiar advantage of binary

over unary representation. In the following lemma we record some of these observations.

4.4 Lemma: Here are the RECW-costs for some functions:

function	single-tape cost	multi-tape cost
selector	linear	constant
test for emptiness	linear	constant
test for equality	quadratic	linear
generalized addfirst	linear	constant
concatenation	quadratic	linear
bitwise addition	quadratic	linear
" multiplication	cubic	quadratic

Proof: Since data access is not free in RECW, we have to rewrite the programs of 2.5 in variable-free form:

empty := if addfirsto then addfirsto else addfirsto $\circ$ c_o^1

c_j^1 := addfirstj $\circ$ epsi (constant function)

epsi := subfirst $\circ$ first (empty word function)

All the functions involved are of linear single-tape cost and of constant multi-tape cost.

eq := if empty $\circ$ s_1^2 then (if empty $\circ$ s_2^2 then c_o^2 else c_{oo}^2)
else if empty $\circ$ s_1^2 then c_{oo}^2 else if equal $\circ$ (first⊗first)
then eq$\circ$(subfirst⊗subfirst) else c_{oo}^2 .

Thus

COST(eq)n = C(n) + COST(eq)(n$\dot{-}$1),

where C(n) is linear (constant) for single-(multi-)tape cost.

We leave the remaining functions to the reader. Q.E.D.

4.5 Now we can show that the compilers of Sect. 3 raise the cost of a program only modestly. The discussion in 4.3 shows that we have to compare Turing machine time and RECW cost; also we have to be faithful to the multi-tape case and the single-tape case.

4.6 Theorem: The compiler in 3.7 from RECW into TURING is quadratic: to any RECW-program A it produces an equivalent Turing machine M s.t., if A is of at least linear cost, then

$$\text{Time}(M) = O(\text{COST}(A)^2) \ .$$

(If A is charged single-tape cost, then M can be single-tape; otherwise M is multi-tape, in fact, has 2 work tapes.)

Proof: Let A be a RECW-program of at least linear cost. Let b be an input on which VAL(A)b is defined. Let n := length(b), m := COST(A)b; thus n=O(m). Let m be the 2-tape Turing machine equivalent to A by theorem 3.7. Started on input b, the machine M simulates exactly the computation steps of VAL(A)b. Each simulation step involves handling of the actual argument(s), which can be done in linear time with the help of tape 2; plus maybe the handling of a term from A, which can be done in constant time. Since M can compute the basis functions of RECW in linear time, the time needed for any simulation step is linear in the length of the actual argument. Each evaluation step of A has cost 1 under multi-tape RECW-cost. Thus there are m evaluation steps, and therefore the length of the actual arguments is $\leq$ n+m=O(m). Thus M uses time O(m) for each simulation step, and

$$\text{Time}(M)n = m \cdot O(m) = O(m^2) = O((\text{COST}(A)n)^2).$$

For the case of single-tape RECW-cost construct M as a single-tape machine. Then the time M uses for each simulation step is quadratic in the length of the actual arguments. But the cost of A for each evaluation step is now linear in the same length. Thus

$$\text{Time}(M)n = O((\text{COST}(A)n)^2).$$

(Note that here we need not, and in general cannot, know the number of evaluation steps.) Q.E.D.

4.7 Theorem: The compiler in 3.9 from TURING into RECW, slightly changed, is quadratic for single-tape cost, and linear for multi-tape cost: to any Turing machine M it produces an equivalent RECW-program A s.t., if M is of at least linear cost, then

$$COST(A) = O(Time(M)^2) \quad (\text{and } = O(Time(M)) \text{ resp.}).$$

Proof: Let M be a k-tape Turing machine consuming at least linear time. Let b be an input on which M terminates. Let n := length(b), m := Time(M)n; then n=O(m). We represent configurations of M by 3k+1-tuples

$$(z, v_1, a_1, w_1, v_2, a_2, w_2, \ldots, v_k, a_k, w_k)$$

where z is the present state of M, a_i is the symbol scanned by the ith head, and v_i and w_i are the content of the ith tape to the left and to the right resp. of the head. We code M's states and tape symbols into a fixed alaphabet in such a way that all code words have the same length q. Let c be the code function extended to words. Then in each step M changes any component of the coded configuration by q symbols, if at all; namely it changes cz, the ca_i's, and the right and left end of the cv_i's and the cw_i's resp. Thus the "next configuration function" g of 3.8 can be programmed in RECW using only functions like subfirst, gaddfirst, and selectors which are of linear/constant single/multi-tape cost by lemma 4.4. (With the usual representation of a configuration as a k+1-tuple we would have to use concatenation of quadratic/linear cost.) The cost for g and also for the halting configuration predicate εH therefore is linear/constant in the length of the configuration. This length is O(n) when M starts to compute, and is increased at each step by at most k·q; thus it will be at most

$$O(n) + m \cdot k \cdot q = O(n) + O(m) = O(m).$$

Thus the cost for the RECW-program in 3.8 of the configuration function f of M is

$$COST(f)n \leq m \cdot O(m) = O(m^2) \quad (\text{and } \leq m \cdot O(1) = O(m) \text{ resp.}).$$

If M is used to compute a word function $h:D \to D$, then we have to account for the cost of the functions read and write, too. Read is a selector, and write involves the identity and constant writing functions; thus both are of linear/constant cost, and do not alter the above result. Q.E.D.

4.8 a) As the compilers themselves, also their cost bound carries over to the more general situations mentioned in 3.11. Thus there are, for example, quadratic compilers between PROBRECW and the probabilistic Turing machines of [Gill 1977].

b) The compilers to and from REC support the "complexity theoretic version of Church's thesis": that all "practical" programming systems are polynomially related, and any problem "arising in praxis" is "computable in praxis" iff it has polynomial complexity.

5. COST ANALYSIS

The simple structure of REC-programs makes their evaluation most transparent, and thus helps to exhibit some basic principles of cost analysis. If so desired, one could transfer these principles to more involved languages. Every programmer knows that the evaluation of a program involves two kinds of work to be done: organization (of data access and control flow), and calculation (evaluation of operations). On a computer often an organization step is faster, and thus cheaper, than a calculation step, and there are trade-offs between the two types of costs. In cost analysis nevertheless, this important distinction either is made by brute force ("we count only multiplications, everything else is free") or is neglected (machine cost). Making use of the parallelism between semantics and cost definition in REC we give a precise definition of "organization" and "calculation", and of their respective costs. Thus we can put different weight on either type of cost, resulting in a different cost analysis. The distinction between organization and calculation also helps to understand probabilistic programs, where part of the organization is "done by chance" in order to save calculations. Our second aim in this section is to start a classification of programs according to their susceptibility to cost analysis.

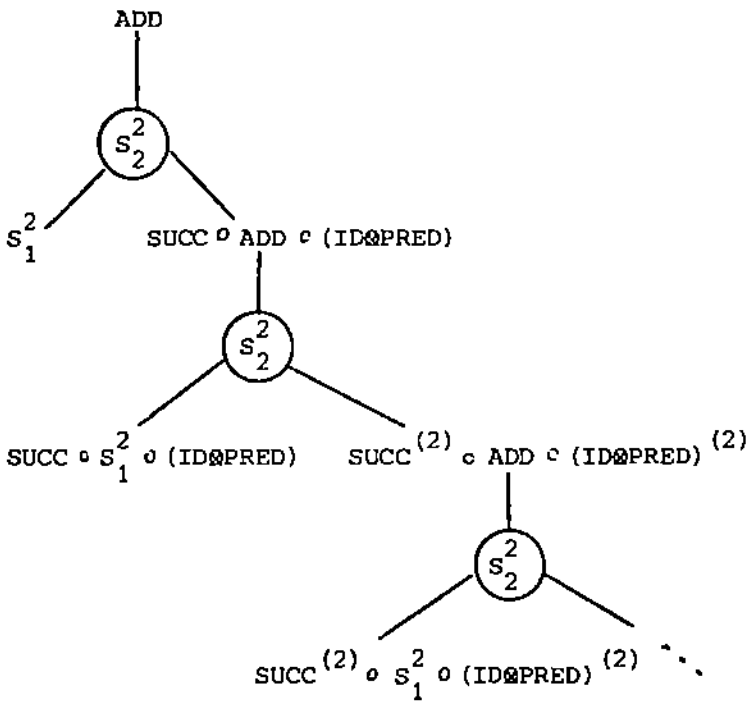
5.1 The evaluation of a REC-program on an input consists of a sequence of tests (if-then-else) and (possibly recursive) calls of function variables. The remaining operators (composition, product, and combination) regulate the data flow. If we follow the flow of control by doing the tests and substituting terms for function variables, and if the program terminates, we get a term which is free of if-then-else and of function variables. This term describes the calculation to be done, including the data access operations. We call it the 'calculation term' CT of the program on that input. We call the sequence of test terms

together with their arguments the 'organization sequence' OS.

5.2 Example: Recall the addition program in 1.1:

ADD:=if S_2^2 then S_1^2 else SUCC ∘ ADD ∘ (ID⊗PRED)

The evaluation of the program on any input can be described by an evaluation tree where vertical lines indicate calls of ADD (the substitution of the defining term for ADD is deleted), and branching is generated by tests (we circle the test predicate).



For any input the sequence of evaluation steps is reflected by a path down the tree. The circled nodes show the evaluated test predicates; together with the respective arguments they yield the organization sequence. The other nodes show the construction of the calculation term; the last node in the path contains the calculation term. Therefore we get:

$CT(ADD,(a,b)) = SUCC^{(b)} \circ S_1^2 \circ (1\otimes PRED)^{(b)}$

$OS(ADD,(a,b)) = ((S_2^2,(a,b)),(S_2^2,(a,b\dot{-}1)),\ldots,(S_2^2,(a,0)))$

where $f^{(b)} := f \circ \ldots \circ f$ (b times).

Another example follows in 5.13.

5.3 Definition: a) Following parallely the definitions of semantics and cost of REC-programs we define the Calculation Term of a program for an input as a recursive function

CT : programs → [terms → [input → terms]]

Since the calculation term will itself be applied to input to yield output, we will write CT(A,b) instead of CT(A)b, to avoid misunderstanding. Similarly we write CT(t,b) instead of CT(A)(t)b in the following equations. Now let $A :\equiv (F_1 := t_1,\ldots, F_k := t_k)$ be a REC-program. Define CT by

(1) $CT(p\circ q,b) := CT(p,VAL(q)b)\circ CT(q,b)$

(2) $CT(p\otimes q,(b,c)) := CT(p,b)\otimes CT(q,c)$

(3) $CT(p,q),b) := (CT(p,b),CT(q,b))$

(4) $CT(\underline{\text{if}}\ s\ \underline{\text{then}}\ p\ \underline{\text{else}}\ q,b) := \begin{cases} CT(p,b);\ VAL(s)b=o \\ CT(q,b);\ VAL(s)b\neq o \\ \text{undefined; o.w.} \end{cases}$

(5) CT(G,b) := G for a function constant G

(6) $CT(F_i,b) := CT(t_i,b)$ for a function variable F_i

(7) $CT(A,b) := CT(F_1,b)$

The Organization Sequence OS(A,b) of a program A for an input b is the sequence of pairs (s,c) evaluated during construction of CT(A,b) where c is an argument, and s is either

(i) a test condition, or

(ii) one of the operators composition, product, combination, test, or call.

(Normally, when we are not accounting for RECW-cost, we will simplify OS by deleting the pairs of type (ii), since their cost is o anyway.) Both CT(A,b) and OS(A,b) are undefined if VAL(A)b is undefined.

b) For a program A which terminates on input b let F:=CT(A,b) be the calculation program CP(A,b) of A on b.

5.4 Theorem: A program and its calculation term yield the same result on any input on which the program terminates:

$$VAL(A)b = VAL(CT(A,b))b$$

Proof: Let A be a program which terminates on the input b. We have to show:

$$VAL(F_1)b = VAL(CT(F_1,b))b$$

For any term t and any argument c we prove

$$VAL(t)c = VAL(CT(t,c))c$$

(This proves especially that CT(t,c) is defined iff VAL(t)c is defined.)

Proof by induction on the number n of steps to compute VAL(t)c.

Beginning: n=1: Then t is a function constant g. Thus CT(g,c)=g, and

$$VAL(CT(t,c))c = VAL(g)c = VAL(t)c$$

Induction step: t cannot be a function constant. Consider the case $t \equiv p \circ q$:

$VAL(CT(p\circ q,c))c$ = (by definition of CT)

$= VAL(CT(p,VAL(q)c) \circ CT(q,c))c$ = (by definition of VAL)

$= VAL(CT(p,VAL(q)c))(VAL(CT(q,c))c)$ = (by induction hypothesis)

$= VAL(CT(p,VAL(q)c))(VAL(q)c)$ = (by induction hypothesis)

$= VAL(p)(VAL(q)c)$ = (by definition of VAL)

$= VAL(p\circ q)c$

The other cases are analogous. Q.E.D.

5.5 Definition: The calculation cost and the organization cost of a program A on an input b are the cost of its calculation term and its organization

sequence resp., on that input:

CALCOST(A)b := COST(CT(A,b))b

ORGCOST(A)b := Σ {COST(s)c ; (s,c) ∈ OS(A,b)}

(where COST(op)c := OPCOST(op)c for short. Recall that summands of this type are o except for RECW-cost.)

5.6 Theorem: The cost of a program consists of calculation cost and organization cost:

COST(A)b = CALCOST(A)b + ORGCOST(A)b

Proof: Parallels the proof to theorem 5.4, and is thus left to the reader.

Q.E.D.

5.7 Remark: a) If t is a term without function variables, then CT(t,b)=t.

b) We can make calculation cost and organization cost depend on the input size, as we did with cost itself, and can do worst case analysis with these concepts. Thus we define for n∈ℕ

CALCOST(A)n := max {CALCOST(A)b ; size(b)=n}

ORGCOST(A)n := max {ORGCOST(A)b ; size(b)=n}

Note that theorem 5.6 does not hold if n is substituted for b. For example, in the VREC-program.

$f(x_1,\ldots,x_n)$:= if x_1=0 then Σx_i else if Σx_i=0 then 0 else 1

for any input of size n either the calculation cost or the organization cost is n-1, but not both. Thus

CALCOST(f)n = ORGCOST(f)n = COST(f)n = n-1.

This is, however, not a typical situation. In examples for arithmetic complexity, normally inputs of the same size even produce the same calculation term and the same organization sequence. We call such a REC-program oblivious, since the Turing machine simulating it by theorem 3.7 is oblivious (i.e. the

head movements depend only on the input size, not on the actual input).

For an oblivious REC-program A we define

$$CT(A,n) := CT(A,b)\ ,\ OS(A,n) := OS(A,b)$$

when b is any input of size n. This yields

$$CALCOST(A)n = COST(CT(A,n)),$$

and from theorem 5.6

$$COST(A)n = CALCOST(A)n + ORGCOST(A)n.$$

5.8 The theorems 5.4+6 show that CT and OS have the desired properties: For a fixed input (size) they allow to break up a program into a calculation part and an organization part. Of course the calculation term is not actually constructed during evaluation; it rather serves to make explicit the actual calculation done for that input (size). Especially for oblivious programs, however, the calculation term can be thought of as a piece of hardware. And for programs like Strassen's matrix multiplication or the Fast Fourier Transform it might pay to wire such circuits for often used sizes.

In the literature, to measure the complexity of concrete problems normally one uses random access (register) machines or straightline programs (case of arithmetic complexity) or Boolean networks (case of Boolean complexity). For the following discussion let us call such devices hardcopy programs. Two approaches can be distinguished in presenting a fast method for solving a problem P. Either for each n one produces a hardcopy program A_n which solves P_n (problem P restricted to size n). The A_n are like calculation programs (def. 5.3.b) of an unwritten 'master program'. Or else one gives an algorithm A, say in ALGOL-notation, and indicates (more or less) how hardcopy programs A_n can be obtained informally from A. In either approach one considers $hn := cost(A_n)$ as an upper bound for the complexity of P. The cost for the organization, namely the second term in theorem 5.6 (in the above terminology)

$$cost(A)n = cost(A_n) + \text{cost to get } A_n \text{ from } n,$$

is not taken into account. Any of the more involved (e.g. recursive) algorithms for concrete problems shows that this is a rather crude way to proceed. An impressive case of the type of trade-off given in theorem 5.6 is the shortest-path-algorithm in [Mahr 1979]. There for any graph a hardcopy program which can be very cheap (in calculations), is constructed by solving an organizational problem which can be very expensive (have high organization cost), namely by finding a separating set.

When somebody brings forth a fast method in the above way, the unwritten program or the unwritten hardcopy programs are implicit in the presentation; thus the organization cost could be recovered in an informal way. The negligence seems to be more serious when one wants to prove, dually, that the function h is a lower complexity bound for the problem P, i.e. in the terminology of REC

(1) $\forall A[A \text{ solves } P \rightarrow \exists^{\omega} n[COST(A)n > hn]]$, or equivalently

(1') $\forall A \; \exists^{\omega} n[CT(A,n) \text{ solves } P \rightarrow COST(A)n > hn]$.

Since such a formulation is impossible with hardcopy programs, in the literature one proves instead the stronger statement

(2) $\exists^{\omega} n \; \forall A_n \; [A_n \text{ solves } P \rightarrow COST(A_n) > hn]$.

Implicit here is of course the idea that the organization cost should never exceed the calculation cost. It is easy, however, to construct for each honest function h a problem P which has a complexity just above h, but is trivial to calculate (i.e. has calculation cost 0 for each input size); thus (1) is true and (2) is false. This shows that the approach suggested in this paper yields a better foundation to concrete complexity than the 'hardcopy way'. Also the 'adaptive' lower and upper bounds of [Mahr 1979] ask for such a framework. In 5.10.d we will discuss another aspect which makes computation programs superior to hardcopy programs.

5.9 Definition: a) A line F:=t in a REC-program calls a line G:=s (or

calls G) if the term t contains the symbol G.

b) A line is recursive if it calls itself.

c) A REC-program is stratified if each line calls only itself or lines below.

d) To compress a stratified program do the following for i=2,...k: If line i is non-recursive, substitute t_i for F_i everywhere in lines 1,...,i-1; remove line i. If line i is recursive and is not called in lines 1,...,i-1, remove it.

5.10 Remarks: a) The evaluation (and thus the cost analysis) of a stratified program is especially simple: A line i can call either itself (recursive) or a line further down (subroutine). In the latter case the call can be completed without use of the lines 1,...,i. Thus any line (together with the lines further down) forms itself a (stratified) REC-program.

b) The compression of a stratified program is stratified. Except possibly for the first one, its lines are recursive. Compressing leaves the evaluation tree (see 5.2) unchanged.

c) The normal form of corollary 3.10 yields to any REC-program an equivalent stratified program. It seems plausible that most REC-programs arising in praxis are stratified (possibly by renumbering the lines).

d) Straight-line programs (turned upside down) correspond to extremely simple REC-programs: each line is non-recursive, and contains a single operation. The input variables are the same; the defined (intermediate and output) variables in the straight-line program are function variables in the REC-program. Compressing such a REC-program yields a single non-recursive line, which is the explicit definition of a function by a term which gave rise to the straight-line program. This line is also the calculation program for that input (size). This shows another advantage of the REC-formalism. Hardcopy programs (see 5.8) are like machine code: cumbersome to write and unintelligible. Instead we can use the full power of the high-level language REC, adapted to the data type

in consideration, without being less precise; we get down to the "machine level" by producing the calculation program, which is a mechanical process.

5.11 Examples: a) Matrix multiplication: Let us write A[ℓ,m] for an ℓ × m-matrix, A(i,j) for its entries. The definition

(1) $\quad C(i,k) := A(i,1)\cdot B(1,k) + \ldots + A(i,m)\cdot B(m,k)$

of the product C[ℓ,n] of two matrices A[ℓ,m] and B[m,n] leads directly to a program, say, in ALGOL or REC. For fixed dimensions the ALGOL-program resolves into the obvious straight-line program; the calculation program of the REC-program is

(2) $\quad MM(A,B)_{i,k} := A(i,1)\cdot B(1,k) + \ldots + A(i,m)\cdot B(m,k)$

which is the same as (1). (See 5.10.d.) If, however one treats matrices not as prefabricated, but as a data structure generalized from 'words', one is lead to an inductive definition using operations like firstrow, subfirstrow, addfirstrow etc. (Compare 2.3.) This suggests a recursive definition of matrix multiplication as given by the following cases:

(i) $\quad A[1,1]\cdot B[1,1] = (A\cdot B)[1,1]$

(ii) $\quad (A[1,m],\ D[1,1])\cdot(B[n,1],\ E[1,1]) = A[1,m]\cdot B[m,1] + D[1,1]\cdot E[1,1])$

(iii) $\quad A[1,m]\cdot(B[m,n],\ E[m,1]) = (A[1,m]\cdot B[m,n],\ A[1,m]\cdot E[m,1])$

(iv) $$\begin{pmatrix} A[\ell,m] \\ D[1,m] \end{pmatrix}\cdot B[m,n] = \begin{pmatrix} A[\ell,m]\cdot B[m,n] \\ D[1,m]\cdot B[m,n] \end{pmatrix}$$

This recursion translates directly into the REC-program

```
MM(X,Y) :- if colnX ≠ rownY then error else
if rownX = 1 then
  if colnY = 1 then
    if colnX = 1 then nat(X)·nat(Y)
    else MM(sublastcolX,sublastrowY) + MM(lastcolX,lastrowX)
  else gaddlastcol(MM(X,sublastcolY),MM(X,lastcolY))
else gaddlastrow(MM(sublastrowX,Y),MM(lastrowX,Y))
```

An easy induction shows that the calculation program of MM is (2) as above.

b) Integer multiplication: The well-known algorithm of Karatsuba which multiplies two n-digit numbers in $O(n^{\log 3})$ steps, is based recursively on the following fact: Any n-digit numbers x and y over the basis B, where n is even, m:=n/2, can be written as

$$x = a \cdot B^m + b,\ y = c \cdot B^m + d$$

with m-digit numbers a,b,c,d. Thus we can multiply

$$x \cdot y = a \cdot c \cdot B^n + (a \cdot c + b \cdot d - (a-b) \cdot (c-d)) \cdot B^m + b \cdot d$$

with 3 (instead of 4) m-digit multiplications, since a·c and b·d need not to be produced twice. The relevant lines in a straight-line program are

u:=a·c, v:=b·d, w1:=a-b, w2:=c-d, w:=w1·w2

If we translate this into a REC-program KMULL as in 5.10.c, we get 3 multiplicaitons as expected. Every evaluation of KMULL however, uses 5 multiplications, since the functions U and V are called twice each. Also the calculation program contains 5 multiplications. Thus we have to be more careful, and write something like

$$KMULT := ADD^3 \circ (SHIFTn \cdot S_1^3, SHIFTm \cdot ADD^3 \circ (ID^2 \otimes MINUS), S_1^3) \circ (U,V,W)$$

where the programs for U,V,W are analogous to above. This program contains, and uses, only 3 multiplications which are piped explicitely to the appropriate locations; the same is true for the calculation programs. It is the strict coupling in REC between semantics and cost which forces us to be this explicit. In contrast, the cost of the straight-line program depends, besides on its semantics, on its implementation: If we compute it top-down (substituting values), we multiply 3 times; if we compute it bottom-up (substituting terms, evaluating at the end), we multiply 5 times.

c) We suggest that the reader defines inductive data structures which fit naturally to the fast matrix multiplication of Strassen and to the Karatsuba algorithm, or to any other recursive algorithm. How does the change of the data structure influence the organization cost? Are recursive algorithms so

hard to implement only because they are strained through a non-inductive data structure?

5.12 Our distinction between calculation and organization (def. 5.3) is tailored to oblivious programs. Often, e.g. in efficient algorithms for sorting and searching, the control depends also on the actual input. In such a situation we divide the basic functions of a REC-program into administering and working operations. We call a term working if it contains a working operation; otherwise administering. Further we call a test administering or working depending on what its condition is. Typically, administering are id, erase, zero, succ, pred ; then data access, index computation, and initialization are administering. We change the definition of the calculation term CT(t,b) for the case t is a working test of the form (if s then p else q), as follows:

CT(t,b) := (if CT(s,b) then CT(p,b) else CT(q,b)).

5.13 Example: Binary search. Consider the following program which searches for an element X in a LIST between the positions $L \leq M$:

SEARCH(LIST,X,L,M) := if L=M then L else if $X \geq$ LIST($\lceil L+M/2 \rceil$)

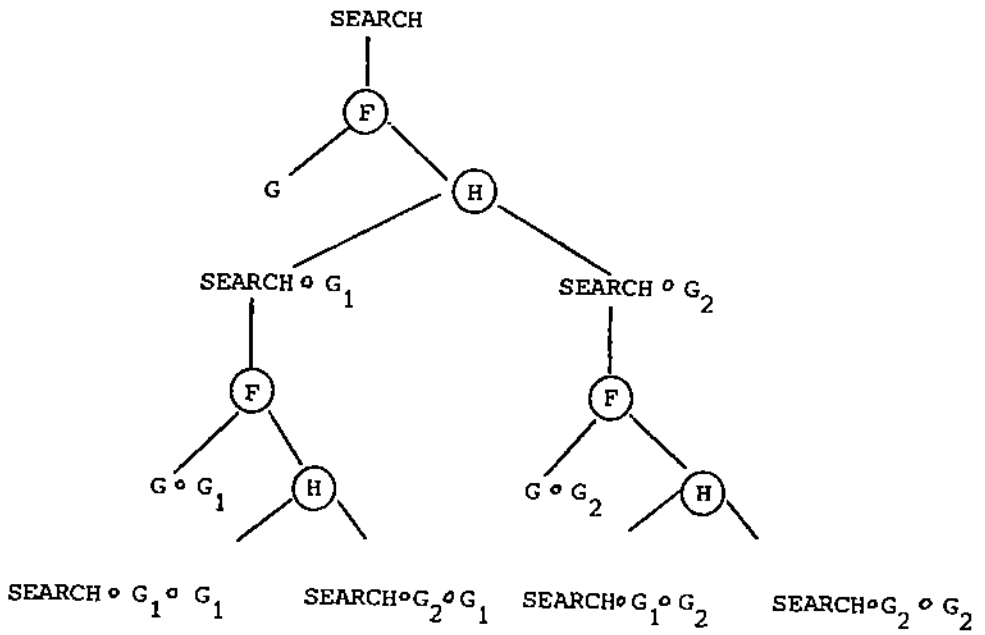
then SEARCH(LIST,X, $\lceil (L+M)/2 \rceil$,M)

else SEARCH(LIST,X,L, $\lceil (L+M)/2 \rceil - 1$)

The variablefree program is of the form

SEARCH;= if F then G else if H then SEARCH $\circ$ G_1 else SEARCH $\circ$ G_2

This yields an evaluation tree of the form



1. case: $\geqq$ is an administering operation. Then for a list of length n there are 2^m calculation terms of the form $G \circ G_{i_1} \circ \ldots \circ G_{i_m}$, $\lfloor \log n \rfloor \leq m \leq \lceil \log n \rceil$, corresponding to the possible locations of X in the list.

2. case: $\geqq$ is a working operation. Then the length of the list determines the calculation term uniquely. It consists of the complete decision tree which yields access to the location of X, e.g. for n=4:

$$\underline{\text{if}}\ H\ \underline{\text{then}}\ (\underline{\text{if}}\ H\ \underline{\text{then}}\ G \circ G_1\ \underline{\text{else}}\ G \circ G_2) \circ G_1$$
$$\underline{\text{else}}\ (\underline{\text{if}}\ H\ \underline{\text{then}}\ G \circ G_1\ \underline{\text{else}}\ G \circ G_2) \circ G_2$$

5.14 Remarks: a) Example 5.13 shows that there is no rigid distinction between calculation and organization. In most practical cases one will have no problem to distinguish, depending on the type of cost one is involved with, administering from working operations. Normally, however, part of the calculation term will be evaluated during the organization phase. To clarify the situation we could split off access terms from the calculation term, containing only administering operations.

b) The basic functions COIN and COINp in the nondeterministic and probabilistic versions of REC (see. 2.11), are always administering. Thus, there the calculation term depends not only on the input, but on the result of COIN as well; it does not contain COIN. This makes apparent how the computation is organized by throwing the coin.

5.15 Definition: Types of REC-programs

a) A line in a REC-program is simple if it is of the form

$$f := \underline{\text{if}}\ p\ \underline{\text{then}}\ s\ \underline{\text{else}}\ t[f\circ h_1,\ldots,f\circ h_m]$$

where (i) there are no other tests and no other occurances of f (esp. no nesting of f),

and (ii) there is a well-founded partial ordering on the data (i.e. no datum admits an infinite descending chain among its ancestors) s.t. in the sense of the ordering the h_i are strictly decreasing, and the set defined by p is closed under ancestors.

b) A simple line is primitive if the partial ordering in condition (ii) is the natural term ordering of the data structure.

c) A REC-program is

- loop-free if it is stratified and has no recursive line;
- a tree (a stick) if it is loop-free and contains working tests (no tests at all);

- tree- (stick-) generating if it is stratified and contains (no) working tests;
- simple (primitive) if it is stratified and all its recursive lines are simple (primitive).

5.16 Remarks: a) The evaluation tree (see 5.2) of a loop-free program is finite. Thus these are only finitely many different calculation terms. To determine the cost for an input one has to go along the corresponding branch; the maximum taken over the corresponding branches yields the cost for an input size. A stick program expanded yields a straight-line program. (See 5.10.d) The evaluation tree of a stick program consists of a single branch; that of a tree-program is a tree alright. The calculation term of a stick program is the same for all inputs. The compression of a loop-free program is a single non-recursive line, i.e. an explicit definition. Such a program does not call at all; one determines its cost by inspecting the defining term. On any input the calculation program of a tree-(stick-) generating program is a single-line tree (stick) program. In the literature sometimes informally the term 'tree program' is used for what corresponds to our tree-generating programs.

b) A program which is primitive over the data structure $< \mathbb{N} ; 0, succ >$, defines a primitive recursive function. It seems plausible that any program arising in praxis is simple, or can be made simple very easily. The four auxiliary functions in the normal form of 3.10 are very 'simple' simple functions. A simple program terminates on all inputs. In many cases its cost function, calculation terms and organization sequences are 'simple', and easy to determine. (See 5.18.)

5.17 With the help of the concepts of this Section we can now describe the

process of cost analysis of a REC-program A. First stratify the program by renumbering lines. (If this is not possible, see below.) Starting at the bottom determine the cost of each line of the program (see remark 5.10.a), substituting the cost functions of the lines which are called upon. The cost of the first line is the cost of the program.

This is normally a simple process if the program is simple. (See 5.15a.) Non-stratifiable programs can often be handled by the same method by figuring out the cost of two 'simultaneous recursions'

F:=s[F,G], G:=t[F,G]

from the cost of the combined line

(F,G):=(s[F,G],t[F,G]).

From the information collected during the cost analysis one determines (if possible) a definition of 'input size' which makes the program oblivious, and thus gets a size-depending cost function. While doing the cost analysis one can also construct the calculation terms and the organization sequences.

5.18 Cost analysis of simple programs: Consider the simple line (def. 5.15.a)

(1) fx := if px then gx else t[x,f(h_1x),...,f(h_mx)]

If the data structure is $\mathbb{N}^n$ for some n, px is 'x_1=0', m=1, h_1x := (x_1-1, x_2...,x_n), then f is defined by primitive recursion from the functions involved. More generally, simultaneous primitive recursion, course-of-value recursion and the nested recursions of Péter [1936] are all simple. From the point of view of foundations our generalization from 'primitive' to 'simple' is not satisfying since the existence of the partial well-ordering is not a syntactical condition. For programming and cost analysis, however, simplicity is as important as in every-day life: it is not chained rigidly to the data structure; it is rather easily checked; most recursive programs in praxis have it. We cannot dispense with the requirements concerning the partial well-ordering. According to an

unpublished result of F. Müller (see [Fleischmann-Müller-Siefkes 1975]) we can still define all recursive functions in RECS if we restrict programs syntactically to the form (1).

By applying def. 4.2 to (1) we get a recursion for the cost of f:

(2) $COST(f)x = COST(p)x + \{\underline{if}\ px\ \underline{then}\ COST(g)x\ \underline{else}$

$$\sum_{i=1}^{m} [COST(h_i)x + COST(f)(h_i x)] + COST(t)(x, f(h_1 x), \ldots, f(h_m x))\}.$$

To simplify the analysis let us assume that we can define an input size which makes the program oblivious. Often then the predicate p depends only on the input size, not on the actual input. Thus we can define a predicate q on $\mathbb{N}$ s.t.

(a) $qn :\Leftrightarrow px$ for some x of size n.

Let us further assume that COST(t) depends only on size(x) instead of on the actual arguments. If we write h for $(h_1, \ldots, h_m)$, we get

(3) $$COST(f)n = COST(q)n + \{\underline{if}\ qn\ \underline{then}\ COST(g)n\ \underline{else}$$
$$COST(h)n + COST(t)n + \sum_{i=1}^{m} COST(f)n_i\}$$

for $n:=size(x)$, $n_i:=size(h_i x)$. Solving (3) is made easier by further assumptions. For example, often $COST(f)(size(h_i x))$ for all i and x depends only on size(x). Then we can define a function $k: \mathbb{N} \to \mathbb{N}$ s.t.

(b) $COST(f)(size(h_i x)) = COST(f)(kn)$

for n:=size(x). By distributing COST(q)n into the if-then-else, and abbreviating

(c) $rn := COST(q)n + COST(g)n$

$sn := COST(q)n + COST(h)n + COST(t)n$

we get

(4) $COST(f)n = \underline{if}\ qn\ \underline{then}\ rn\ \underline{else}\ sn + m \cdot COST(f)(kn)$.

In many cases this recursion is easily solvable. Since p is a terminating condition, it is plausible that r can be bounded by a constant c in the range of q. We quote two known solutions to (4): Let sn be bounded by a polynomial of degree b. If

(d) $qn :\equiv n=0$, $kn := n \dot{-} 1$,

then for $\lg n := \log_m n$

$$(5)\quad \mathrm{COST}(f)n = \begin{cases} O(m^{n+b\cdot \lg n}) & ; m > 1 \\ O(n^{b+1}) & ; m = 1 \end{cases}$$

If for some $c > 1$

(e) $qn :\equiv n=1,\ kn := \lceil n/c \rceil$

then for $d := \log_c m$

$$(b)\quad \mathrm{COST}(f)n = \begin{cases} O(n^{d}) & ; d>b \\ O(n^{b}\cdot \log n) & ; d=b \\ O(n^{b}) & ; d<b \end{cases}$$

6. SEMANTICS AND COST OF PROGRAMS

We conclude the paper by collecting a couple of properties of REC, and commenting on the underlying ideas. We start by supplementing 1.6 with the straightforward proof that the semantics of REC is essentially deterministic (6.1+2). From this basis, in 6.3 we justify the definitions of semantics and cost for REC, and for its variants of Sect. 2. In 6.4 we clarify the discussion of 6.3 by commenting on the concept of nondeterminism. In the central part 6.5 of this section we distinguish a semantic and a syntactic way to give a meaning to a set of recursive equations: Dedekind-Skolem-Herbrand-Gödel vs. Thue-Post-Gödel-Church-Kleene-Turing. We claim that in REC the two ways converge, allowing to compute "naturally". We support this bold claim in 6.6 by touching on the method of algebraic specification, maintaining that it is important to keep apart a program and the underlying data structure. In 6.7+8 we prove a theorem of Büchi, quoted in 6.5, that exactly the hyperarithmetic functions are Herbrand functions. In 6.9 we collect the points made in 6.3 and 6.5, showing that REC is a functional language in the sense of Backus [1978], and extending these merits to the cost definition for REC. Consequently, in 6.10 we compare REC and the programming systems of Backus. We conclude the paper in 6.11 with a general remark: how would life change if we would distinguish properly between "administering" and "working"?

6.1 Theorem: REC has a hyper-strong Church-Rosser property; namely given any two evaluation sequences of the same program on the same input, either both do not terminate, or both terminate yielding the same result and containing the same steps (though not necessarily in the same order).

Proof: Let t be a term with input b. Let R and S be two sequences computing VAL(t)b, of length m and n resp.

Show: If m is finite, then m=n, and R and S contain the same steps and yield the same result. (The theorem follows by symmetry from def. 1.5)

Proof: Induction on m for all t, b, R, S. If m=1, then t is a function constant, and R=S. Thus let m>1. If t is a composition, a test, or a function variable (clauses (1), (4), (6) in def. 1.5), then R and S have the same second line. For this line and their evaluation sequences the result holds by induction hypothesis. Thus it holds also for t, b, R, S.

Thus let t: ≡ (p,q) be a combination. (The case of product (p⊗q) is similar.) By clause (3) of def. 1.5, on the second line S starts by evaluating either p or q on b. Let it be the latter (the other case being symmetric). Let V be the subsequence of S computing VAL(q)b, of length j. Since R terminates, it must evaluate q on b, too (though not necessarily first). Let U be the corresponding subsequence of R, of length i. Since i < m, by induction hypothesis j = i, and U and V contain the same steps and yield the same result. Now, since V is finite, S must evaluate p on b, too (starting at latest when V is finished). By the same argument, again R and S do the same job on VAL(p)b, and thus on VAL(t)b. Q.E.D.

6.2 Corollary: For any REC-program A and any input b, COST(A)b does not depend on a strategy for evaluating A on b.

6.3 Thus theorem 6.1 justifies the cost definition for REC: the cost of a program depends only on the semantics, and not in a hidden way on the implementation of the language, i.e. on anyone's interpretation of the semantics definition. Of course, the implementation dictates the operation cost, and thus explicitly influences the cost definition.

Theorem 6.1 also justifies the semantics definiton for REC. Namely one might object that the recursive definition of VAL is not adequate to explain the meaning of programs which are themselves recursive definition. Def. 1.5., however, is a single, simple recursion (though not "simple" in the sense of

def. 5.14.b), which, according to th. 6.1, is immune against the user's intuition about recursion. The definition of VAL is self-explanatory, and serves to evaluate arbitrarily complicated recursive programs. Thus VAL functions as well as (or better than) any interpreter written in its own language: once understood it can be used for any program.

The definition of the semantics of a language ought to, and often will, mirror the pecularities of the language itself. Any semantics of a logic language (implicitly or explicitly) uses words like 'and', 'not', 'for all' in the metalanguage to define and justify their use in the object language. In the same vein the semantics of REC is given by a deterministic recursion. We could make def. 5.1 completely deterministic by requiring that in clauses (2) and (3) p is always evaluated first. Or else we could implement VAL on several processors and create PARREC (see 2.11.a), allowing parallel processing (bounded or unbounded, depending on whether we use a fixed or an unbounded number of processors). This is a decision on the implementation level which does not affect the semantics. Thus the slight indeterminacy in the definition of VAL points to a degree of freedom in the implementation, but not to a nondeterminism in the language. If on the other hand we allow nondeterminism in the language by adding COIN (see 2.11.b), then the semantics itself becomes inherently nondeterministic by the added clause for COIN. By adding a probabilistic choice COINp instead (see 2.11.c) we switch from absolute to probabilistic statements. Stated precisely the clause for COINp is

$$\text{Prob}\{\text{VAL}(\text{COIN}p)\varepsilon := c\} = \begin{cases} p \ ; \ c = 0 \\ 1-p \ ; \ c = 1 \end{cases}$$

Thus in PROBREC the semantics of a program is given through a statement on probabilities.

6.4 At least in our macroscopic world every event is determined, whether it happens with or without our interference. Often, however, it is impossible or too costly or irrelevant for us to obtain enough information beforehand, or to determine the event from it. Then we call the event un-(or under-) determined. We call a program which allows undetermined steps, as well as its computation 'nondeterministic'; although the latter should properly be called 'undetermined' (sc. by the program and the input). The evaluation of a nondeterministic program on a given input can be depicted by an "evaluation tree" where each branching represents the possible choices at an undetermined step. (Such evaluation trees should not be confused with the ones introduced in 5.2, which picture the evaluation of deterministic REC-programs, where the branching corresponds to the possible outcomes of tests on different inputs.) We supplement the discussion in 6.3 by distinguishing several cases of nondeterminism:

(i) Parallel computation: Several subgoals have to be achieved; the order is irrelevant and thus not specified by the program. Here the evaluation has to cover all branches of the evaluation tree. If the computation facilities allow it and the subcomputations do not disturb each other, one can do parallel processing; thereby saving time, but normally not steps. Otherwise one can make the computation deterministic by picking the next step according to a strategy.

(ii) Computation through derivation systems: The possible computation steps are specified by rules; either the result of their application or the choice which one to apply, is undetermined. Normally only one or a few of the

infinitely many possible computations lead to a meaningful result. One can make such systems deterministic by enumerating derivation sequences, which is an unnatural and costly way. Normally the number of choices is decreased by choosing steps according to a strategy.

(iii) Search problems: Programs searching in a finite set are the practical brothers of derivation systems. Examples are the search for a Hamiltonian path of a graph, or for a satisfying evaluation of a propositional formula. As a rule such programs consist of an easy nondeterministic description of a solution. This skeleton is either made deterministic by some strategy, or used for a probabilistic program.

(iv) Asynchronous computations: Several processes work in parallel, they call each other, and shared resources. They may in themselves work deterministically, but the system cannot (or will not) forsee their decisions. Thus this is not a nondeterministic program, but an undetermined system. (See 2.11.d.)

6.5 Anyone given the recursive definition of multiplication in I.1 and asked to multiply 2 by 2, will figure out what the definition "means": he will come up with the same result as anybody else, and even with the same computation steps (in some order). This is true for all primitive recursive, and even for all simple recursive (def. 5.14.b) definitions: the syntactic form of the program implies how to "compute naturally" with the programs, namely

(i) "figure out" values as far as possible (in REC: evaluate control operations and basic functions);

(ii) "call an equation" by passing an argument to it.

Since any call (f,a) fits exactly one equation, the strategy is unique up to the order of the calls. Evaluation strategies were therefore no issue in the early work on recursive definitions (Dedekind, Peano, Skolem). Dedekind had proved that every primitive recursive definition has a 'meaning', namely a

unique total function as solution. Presumably influenced by Cantor's constructive attack at the continuum, he generates this solution as the limit of the sequence of partial solutions which the primitive recursive definition, used as an operator, creates when started with the empty function. In this way he replaces the recursive definition by an explicit one. For his incompleteness theorem [1931] Gödel formalized this step in the β-function.

When Ackermann and Péter found recursive definitions of functions which are not primitive recursive, the search was on for a general definition of 'recursive function'. Herbrand proposed (see Gödel [1934]) to use Dedekind's theorem as a definition: the Herbrand functions are those which are the unique total solution of a system of (recursive) equations. His definition fails, since Herbrand functions need not be computable (see 6.7+8 below) Apparently, Gödel realized this; in [1934] he changed the definition as follows. Call a 'derivation sequence' any sequence of steps, performed on a system of equations, of either of the forms

(iii) substitute a value for a variable;

(iv) replace a variable-free term fa by the value b if the equation fa = b is already derived.

Gödel called a function recursive if there is a set of equations such that for every argument every terminating derivation sequence yields the value of the function. Kleene [1936] investigated this notion generalized to partial functions; see the book Kleene [1952]. Herbrand's definition is generalized by calling a partial function Herbrand if it is the unique maximal (in the sense of $\subseteq$) solution of a set of equations.

Gödel's definition is syntactic: since it seems no longer obvious how to compute with arbitrary sets of equations, the informal process of computing is replaced by applications of two formal rules. Derivation systems for computations had been introduced by Thue in [1914], and studied by Post in [1921]. Besides Gödel, Post (in his "production

systems" of [1936] and [1943]) as well as Church and Kleene (in the λ-calculus, see e.g. [Church 1941]) used derivation systems to define the recursive functions. Post's analysis of 'computing' in [1936] shows best that machines, as e.g. Turing's, are nothing but derivation systems made deterministic. Likely Gödel saw all this; he was reluctant to accept the Church-Post-Turing thesis that production systems are enough to model 'computing'. This seems the wiser as we still do not understand how in nature a "production system" reproduces and changes itself. Possibly recursions of higher order are involved, which embody infinite computations into single steps. (See the work on the hyperarithmetic hierarchy, as described e.g. in Sect. 16.5 of [Rogers 1967]. See also the books by Eigen and Schuster [1979] and Hofstadter [1979].) Today research on the evaluation of recursive functions concentrates on the syntactic approach.

The above method of Cantor-Dedekind-Gödel-Herbrand to semantically assign a meaning to a set of recursive equations, revived in one of Kleene's recursion theorems (thm. XXVI, §66, in [1952]), and led to the lattice-theoretic setting of Scott and Strachey (see e.g. [1971] and Stoy [1975]). There a (partial) function is recursive iff it is the least fixed point (which is unique) of a set of equations.

Thus, there seems to be a perfect analogy between logic languages (as the predicate calculus) and programming languages. Programs and the well-formed formulas of logic are both described grammatically. A semantics function determines which formulas are meant to be true or false; just as the fixed point semantics assigns a recursive function as meaning to a set of equations. Derivation systems translate these semantic concepts into syntactic ones, in both logic and programming. But actually the distinction between syntax and semantics is not at all clear in the area of programming. There is no "semantics" of programs which does not provide for an effective way of computing the meaning of a program through a series

of steps. For example, one proves that the least fixed point of a set of equations exists uniquely by approximating it effectively by partial solutions, as described above for primitive recursive functions; for arbitrary recursions the evaluation strategy dictates the approximation, and thus the fixed point. The "semantics" definition of REC seems, at second glance, syntactic: the evaluation strategy is given as a (though nearly deterministic) derivation system. The definition of VAL, however, follows the grammatical structure of the program, exactly as in logic it follows the grammatical structure of the formula. In this sense def. 1.5 carries out the old task of "computing in a natural way" not only the primitive recursive functions, but all recursive ones. Syntax and semantics are unified. Derivation systems when made deterministic by brute force, as e.g. in Markoff algorithms, are ugly and expensive. REC instead simply computes deterministically following the term structure, and thus observes that even on the highest level computing "means" computing.

When "programming with equations" (as e.g. in [Hoffmann - O'Donnell 1979]) or even with the full predicate calculus (see e.g. [Kowalski 1979]), one exploits the structure of logic systems for programming. Thus, the semantics of a system of equations is the set of all equations which follow logically from it; one can compute using a logic derivation system. To single out syntactically, however, those systems of equations which define a function on a given data structure, one has to resort - as it seems - to computation steps.

Since he built up terms by arithmetic operations only, in his definition of recursive functions Gödel needed equations with compound left sides, as well as several equations for the same function, to allow definition by cases. As a result it is undecidable which sets of equations are "programs" (define a function recursively) - a fact which is hidden by a trick in Kleene's "enumeration theorem" (Kleene [1943]). With the help of the test operator (the other operators are implicit

in Gödel's term definition) we restrict sets of equations syntactically to the simple form of REC-programs, which naturally allow deterministic computations. The operators are taken from the book [Brainerd-Landweber 1974]; there, however, recursive functions are defined via iteration or minimization, not through a general recursion schema. As mentioned in 5.17, F. Müller in [Fleischmann-Müller-Siefkes 1975] uses a special recursion schema to define the recursive functions. From this the REC-formalism evolved. In his book [1976], Bird gives two definitions of the semantics of recursive equations, one of which ("call-by-value strategy") corresponds to the definition of VAL. His system is different from REC, however, in that (it does not work with vector functions and) for evaluating it does not follow the syntax of the defining term, and thus the choice of a deterministic semantics seems arbitrary.

Kleene in his book [1952] calls Gödel's definition of 'recursive function' "a bold generalization" from the primitive recursive case. In the light of the REC-formalism it seems rather a bold self-restriction to formal computation systems, for want of an appropriate generalization of the notion of computing. Actually both, primitive recursion and definition through the μ-operator (Kleene [1936]), are special cases of the recursive definitions allowed in REC (cf. Lemma 3.4).

6.6 When "specifying data structures algebraically" (see e.g. [Goguen - Thatcher - Wagner - Wright 1977]) it is convenient to write down as many properties of an operation as one needs, without combining them into a definition, and even without worrying whether the description is unique outside the domain one is working in. There is no sharp boundary between programs and (specification of) data structures; nevertheless, it is important to keep the two distinct, as is done in logic with "inductive" and "recursive" definitions. (The situation is different with recursive and

nonrecursive functions; but similar with programs which can, or cannot, be executed in praxis.) In a given area the data structure should be the common secure ground for any program coming by. This does not mean that a program does not need its own peculiar data structure. Data representations, however, are used very often, thus should be evaluated as fast and reliably as possible, using any tricks. Thus nondeterministic specification might be useful to design a data structure; for practical use it will have to be refined to a deterministic evaluation. Understandable programs over efficient data structures might be a good aim.

To consider an example, Henderson and Morris in [1976] present a nondeterministic LISP-evaluator, arguing that it is often important, e.g. when one works with "streams" (= infinite I/O), to postpone evaluation steps. To us, it seems more appropriate to first define 'streams' abstractly with such examples in mind, and then to use them as a handy data structure.

Actually, algebraic specification and recursive computation complement each other. In the former one defines data representations inductively, equating representations by equations. There seems to be no other way to introduce a data structure, if one does not work with a domain of prefabricated data. Recursion then "unwinds" such data representations, and thus is the natural and most efficient way to compute with them. Linear control structures, as do- and while-loops, iteration, and the μ-operator, are useful in special cases, but are in use mainly as the remains of a time where computing was done solely in the natural numbers. It is high time to divest them from their exclusive rôle in teaching, doing and thinking about computing.

6.7 As mentioned in the beginning of Sect. 6.5, there are non-recursive Herbrand functions. This was shown by Kalmár in [1955]. At about the same time Büchi proved independently the much stronger statement that exactly the hyperarithmetic functions are Herbrand (unpublished, see Büchi [1957]).

In the following we present Büchi's result, formulated for the REC-formalism. We start by giving a RECS-program (def. 1.7) whose unique total solution is non-recursive. To do this we assume any standard coding of Turing machine configurations onto natural numbers be given. Let End be the set of (codes of) end configurations; let Tucc be the "Turing machine successor function", which yields to any configuration the next one (and is the identity on End). End and Tucc are primitive recursive, thus are the unique total solutions of appropriate RECS-programs. Let A consist of these programs with the additional first line

Gx := if $x \in$ End then 1 else if (G∘Tucc)x > 0 then (succ∘G∘Tucc)x else (G∘Tucc)x

Let Halt (Haltj) be the set of configurations which halt (after j steps): i.e.

Halto := End, Halt(j+1) := { $x \notin$ Haltj; Tucc $x \in$ Halt j }, Halt := $\bigcup$ Haltj.

Let g be a maximal solution of A. We want to show that

$$gx = \begin{cases} j+1 & ; \ x \in \text{Haltj} \\ 0 & ; \ x \notin \text{Halt} \end{cases}$$

Thus g is unique (and total). If g were recursive, then the halting problem were decidable, since Halt = sg∘g.
Now we prove by induction on n

$$x \in \text{Haltn} \rightarrow gx = n+1 \ .$$

By definition of A, gx = 1 for $x \in$ Halto. Thus let n > 0, $x \in$ Haltn, y := Tuccx. Then $y \in$ Halt(n-1); therefore by induction hypothesis gy = n > 0. Thus we have $x \notin$ End, g(Tucc x) > 0, which implies from A

$$gx = g(\text{Tucc } x) + 1 = n+1,$$

which ends the induction. Now let $x \notin$ Halt; assume $g\,x \neq 0$. The definition of A yields for y := Tucc x

(a) if gy > 0, then gx = gy + 1 > 0

(b) if gy = 0, then gx = gy = 0

Thus let y_0 := x, y_{j+1} := Tucc y_j. By (b) $gy_j \neq 0$ for all j. Thus by (a) $gy_0 > gy_1 > \ldots$ is a descending sequence of positive integers; contradiction.

It is easy to see that the REC-semantics of the program A is the partial function

$$hx := \begin{cases} j+1 \ ; \ x \in \text{Halt}j \\ \text{undefined} \ ; \ x \notin \text{Halt} \end{cases}$$

i.e. the Turing time (+1). By writing the definition of g a little more carefully as

$$gx = \begin{cases} \min \{j+1 \ ; \ x \in \text{Halt}j\} \ ; \ (\exists n)\ x \in \text{Halt}n \\ 0 \qquad\qquad\qquad\qquad\quad ; \ \text{o.w} \end{cases}$$

we see that (as a relation) $g \in \Sigma_1 - \Pi_1$ of the arithmetic hierarchy. (One builds up the arithmetic relations by applying number quantifiers to recursive predicates; one gets the hyperarithmetic relations by closing under recursively defined universal functions as well. See [Kleene 1943] and [Rogers 1967], ch. 16.) Thus one proves the first half of thm. 6.8 by iterating the foregoing construction through all of the hyperarithmetic hierarchy, starting with a standard universal function instead of with Turing machines.

The other direction of thm. 6.8 is proved by recalling that a relation is hyperarithmetic iff it can be defined in both forms, $(\exists X)P$ and $(\forall X)Q$, where X is a set variable and P and Q are (hyper)arithmetic predicates. (This statement, and its proof, is analogous to the fact that a relation is recursive iff both it and its complement are recursively enumerable.) Thus let A be a REC-program defining the Herbrand function f; let $F := (F_1, \ldots, F_k)$ be the function variables of A. Since A admits a unique maximal solution for all the variables F, we can write

$fx = y \iff \exists F[F$ are functions satisfying A and $F_1\ x = y]$

and

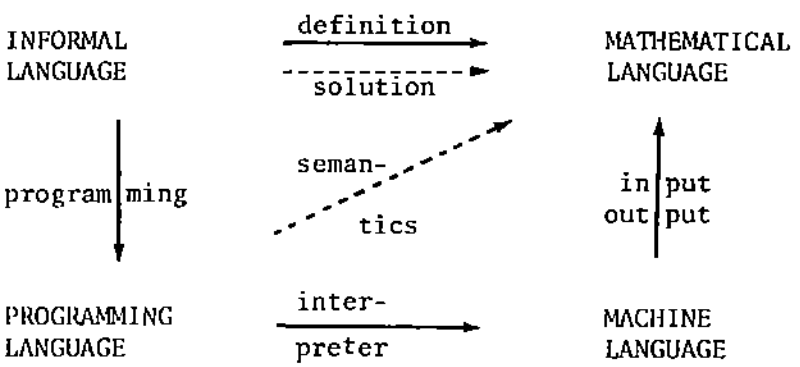
$fx \iff \forall F[F$ are functions satisfying A and $x \in \text{dom}(F_1) \Rightarrow F_1\ x = y]$

6.8 Theorem (Büchi [1957]): Exactly the hyperarithmetic functions are Herbrand.

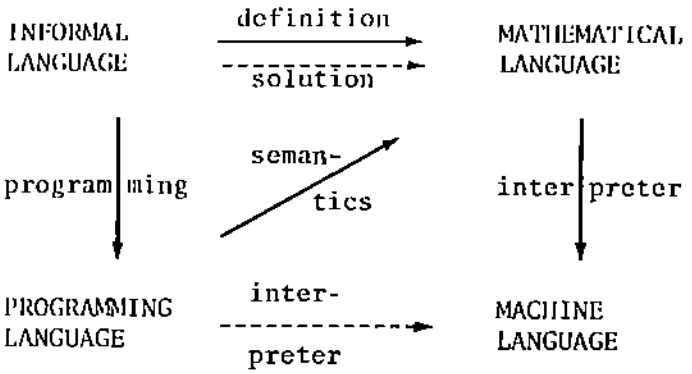
6.9 Usually one defines the semantics of a programming language via a machine, or at least with a machine in mind. A 'machine' need not be greasy or electronical; we think of any device which computes with a prefabricated data structure. The information is stored in locations, which, while

computing, the machine accesses through addresses or some other type of names. One translates a problem given in an informal language, first into the mathematical domain of sets and functions by a precise definition, then into a programming language. A machine runs the program using an interpreter, and thus by an I/O-device provides a solution. If the machine plays its part well, then the program is correct iff the solution coincides with the definition. Thus one gives the semantics of the programming language not directly as a mapping from one formal language into another, but via the detour through the iron teeth of a machine, and for whole programs only. This makes programs so hard to work and to understand. "Algebraization" of machines makes the situation worse: it blurs the difference between computing in a structured or nonstructured way, by clearing the difference away instead of up. We show the situation in the following two diagrams. Solid lines represent the connections described above; the dashed lines, those which are thereby defined.

(1) <u>as it is</u>



(2) as it should be



In the second diagram one provides the semantics directly, by defining the meaning of the basic constructs of the programming language. This induces the meaning of whole programs within the mathematical realm. As a second step one realizes the mathematical equivalents of the basic constructs of the programming language in the machine. Again, this induces a machine interpreter for whole programs. Thus the semantics is in the center of the picture, common base and separating line for the two triangles which join the two formal languages, programming and mathematics, through a vague English on one side and through a rigid machine on the other.

Thus the underlying principle of the REC-formalism is very simple: Compute naturally. The objects to compute with have to be generated by finite means from a finite ground. So a data structure unfolds as (the quotient of) a term algebra. When computing one uses only the tools used to represent the data: One starts with the basic functions, builds new functions from old ones by sequentialization, parallelization, and branching (local principles) and by recursion (the global principle corresponding to the inductive definition of the term algebra). Let us sum up the features of REC, which is a computation formalism structured according to this principle:

(i) Programs have a clear meaning, are easy to write and to understand.

(ii) Programs resemble structurally the functions they define. There are no variables, thus no unstructured data, and no hidden access to them.

(iii) The programming constructs are simple and clearly visible.

(iv) Program construction and verification have nothing to do with machines.

(v) For any data structure there is a closely fitting computation formalism. This yields simple compilers to other computation systems.

The semantics of REC suggests itself from the way we perceive the language; similarly the recursive definition of 'cost' lends itself from the semantics definition, leaving no choice. The cost analysis traces the steps of the computation, as the computation steps trace the definition of the function computed. We can vary the cost by working with a different data structure, or by imagining a different machine implementation. This does not, however, change the concept of cost. For example, the theorems 4.6+7 show that RECW-cost resembles Turing machine time. By simply replacing 'plus' by 'max' at appropriate places, we would depict space instead.

The diagrams (1) and (2) above describe the situation of 'cost' instead of 'semantics' if we just replace the labels 'semantics', 'input, output', and 'interpreter' by 'cost'. Similarly the features of 'semantics' listed above give rise to the following features of 'cost' in the REC-formalism:

(i) The cost analysis is easy and natural.

(ii) The cost analysis follows directly the computation. There are no variables; thus the manipulation of data is clearly visible: there are no hidden costs.

(iii) The organization of the computation lays open, again there are no hidden costs. Therefore some of the basic concepts of cost analysis are easily obtained, as the distinction between calculation and organization, or between different types of programs.

(iv) Cost analysis has nothing to do with machines; only the operation costs stem from an implementation.

(v) For any data structure there is an appropriate version of 'cost'.

Easy compilers support the complexity theoretic version of Church's Thesis.

6.10 Although most of the content of Sections 1 to 4 is contained in [Fleischmann-Mahr-Siefkes 1977] and is independent of [Backus 1978], that paper has influenced our presentation and some of Sect. 6 considerably. Backus has shifted our attention from the cost analysis to include the general question of semantics. His Functional Programming (FP) Systems lift programming a fortunate mile away from traditional programming languages. They are similar to REC-systems, only seemingly more general; they bear the same impact on the situation of semantics. Backus does not distinguish between program and data structure, and does not consider the cost of programs. This allows him to use infinitely many and rather complex primitive functions. So he can multiply arbitrarily large matrices without using loops or recursion. REC is more modest, and thus more accurate. In REC the user has to choose the right data structure, and to spell out his wishes, getting aware of their cost. Sect. 12 of Backus' paper shows that also for him recursion is the most important tool in programming. (Incidently, his recursion theorem 12.5 is the basis for the earlier version of REC in [Fleischmann-Müller-Siefkes 1975].)

We agree less with the second half of Backus' paper. Formal Functional Programming (FFP) Systems are FP-systems plus their formalized semantics, blurred into one system. So the metacomposition rule in the FFP-semantics allows both, to create new FP-control structures and "to write recursive functions without a definition" (p. 633); which is convenient, but does not promote functional programming. Much of the von-Neumann spirit thrown out in FP-systems, comes back into FFP, especially through 'storing and fetching'. On a third level, Applicative State Transition Systems (AST) formalize the implementation of FFP-systems. An AST-system stores only its

own program and the program of the FFP-system it runs presently. Thus its state transitions are simple: changes from one program to another. Implicit in any such transition however are the state transitions of the FFP- run, which in turn code the FP-evaluation. On the top level finally sits a machine which runs the whole hierarchy.

The difference in opinion seems to root in 'storing and fetching'. A program describes a function which changes data. Ultimately we have to store these data somewhere in the physical space. The axiom of functional programming, on which we agree, is: Keep your programming language flexible, and apart from the rigid 3-(or less-) dimensional space. Von-Neumann languages think in the prefabricated data structures of machines. Backus greatly clarifies and loosens this stiff connection by structuring it hierarchily. He does not cut the rope, since he wants to have the AST-system change its programs; thereby turning programs into data, violating his own axiom. A REC-program can store data only "in the data structure". To store them concretely, and to execute programs on them, is a matter of implementation. Therefore problems, as how to store computed values and whether to compute in parallel, do not appear in the semantics. This leaves the language so simple that it is easy to use, and easy to implement on any machine. Thus there is no need to formalize the implementations, or to change programs by the system. Only machine programs have to be changed during computation. If one wants to manipulate a REC-program, one has to choose the right data structure, e.g. RECW, but not to extend the language. To formalize (part of) the semantics of a language in the same language, was a necessary step for Gödel in [1934]; the science of computation still lives from it. In Backus' paper the same step clouds the fine picture of functional programming.

Term algebras are trees; therefore data structure are collapsed trees. The simplest such structure is the full binary tree. When you turn a tree

upside down, it changes into a root system. From such roots grow the trees of recursive programs. If you would not keep its ground-covered roots separate from its sun-hungry branches, a tree would die and decay.

6.11 Our everyday life in universities and elsewhere is split into "administering" and "working". We suffer from the ever more looming overhead of organization. In a today's university the amount of work put into teaching, research and other services shrinks relative to the efforts in trying to organize these activities. E.F. Schuhmacher in [1974] estimates that only 3% of the population of England are engaged in "productive" work (where he does not include education, though). Since administration always produces more administration, it is our task as scientists to find structures which support a trade-off in favor of creative work. Sect. 5 of this paper is a tiny step in this direction: it shows how to distinguish administering and working parts in a program. The same distinction can be made in constructing computers or databases, and in "computerizing" a domain, like e.g. a library. The impact of computers on human life would change, if computer scientists would start to learn that with more organization problems can increase instead of vanish.

REFERENCES

1. Backus, J., 1978: Can Programming be Liberated from the von-Neumann Style? A Functional Style and Its Algebra of Programs. Comm. ACM vol. 21, pp. 613-641.

2. Bird, R. 1976: Programs and machines. Wiley.

3. Brainerd, W., Landweber, L., 1974: Theory of Computation. Wiley.

4. Büchi, J.R. 1957: Letter to, and answer from, K. Gödel.

5. Church, A. 1941: The calculi of lambda-conversion. Annals of Math. Studies no. 6, Princeton Univ. Press, 77 pp.

6. Eigen, M., Schuster, P. 1979: The Hypercycle. A principle of Natural Self-Organization. Springer-Verlag.

7. Fleischmann, K., Mahr, B., Siefkes, D. 1977: Algorithmentheorie. Lect. Notes FB. Informatik, Techn. Univ. Berlin, 1977/78/79.

8. Fleischmann, K., Müller, F., Siefkes, D. 1975: Theorie der Berechenbarkeit. Lecture Notes FB. Informatik, Techn. Univ. Berlin, 1975/76 and 76/77.

9. Gill, J., 1977: Computational Complexity of Probabilistic Turing Machines. SIAM J. Computing, vol. 6, pp. 675-695.

10. Gödel, K. 1931: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte Mathematik Physik vol. 37, pp. 349-360.

11. Gödel, K. 1934: On Undecidable Propositions of Formal Mathematical Systems. Lect. Notes Institute Advanced Study, Princeton, 28 pp.

12. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B. 1977: Initial Algebra Semantics and Continuous Algebras. Journal ACM vol. 24, pp. 68-95.

13. Henderson, P., Morris, J.H. Jr. 1976: A lazy evaluator. Proc. 3rd ACM Symp. Principles Progr. Languages Atlanta, pp. 95-103.

14. Hoffman, C.M., O'Donnell, M.J. 1979: Programming with Equations. Tech. Report Comp. Sc. Dept., Purdue University, 36 pp.

15. Hofstadter, D.R. 1979: Gödel, Escher, Bach: On Eternal Golden Braid. Basic Books, New York.

16. Kalmár, L. 1955: Über ein Problem, betreffend die Definition des Begriffes der allgemein-rekursiven Funktion. Zeifschrift Math. Logik Grundl. Math. vol. 1, pp. 93-95.

17. Kirchner, W., Röhrich, T., Siefkes, D. 1979: Synchronizing Computations by Conditions. Extended Abstract, FB. Informatik, Techn. Univ. Berlin, 9 pp.

18. Kleene, S.C., 1936: General Recursive Functions of Natural Numbers. Math. Ann. vol. 112, pp. 727-742.

19. Kleene, S.C. 1943: Recursive predicates and quantifiers. Transactions ACM vol. 53, pp. 41-73.

20. Kleene, S.C. 1952: Introduction to Metamathematics. Wolters-Noordhoff and North-Holland.

21. Kowalski, R.A. 1979: Algorithm = Logic + Control. Communications ACM vol. 22, pp. 424-436.

22. Mahr, B. 1979: Algebraische Komplexität des allgemeinen Wegeproblems in Graphen. Bericht Nr. 79-14, FB. Informatik, Techn. Univ. Berlin, 133 pp.

23. Péter, R. 1936: Über die mehrfache Rekursion. Math. Annalen vol. 113, pp. 489-527.

24. Post, E.L. 1921: Introduction to a general theory of elementary propositions. Am. Journ. Math. vol. 43, pp. 163-185.

25. Post, E.L. 1936: Finite combinatory processes - formulation I. Journ. Symb. Logic vol. 1, pp. 103-105.

26. Post, E.L. 1943: Formal reductions of the general combinatorial decision problem. Am. Journal Math. vol. 65, pp. 197-215.

27. Rogers, H. 1967: Theory of Recursive Functions and Effective Computability. McGraw-Hill.

28. Schuhmacher, E.F. 1974: Small is Beautiful. Abacus, London.

29. Scott, D., Strachey, C. 1971: Toward a mathematical semantics for computer languages. PRG-6, Oxford University.

30. Siefkes, D., 1979: Recursive Equations as a Programming System. Proc. Frege-Konferenz May 1979, Jena DDR, 16 pp.; to appear.

31. Stoy, J. 1975: Mathematical semantics of programming languages. Project MAC, M.I.T.

32. Thue, A. 1914: Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. Skrifter Videnskapsselskapet Kristiania I, Mat.-nat. Klasse, no. 10, 34 pp.